Spring 5-15-2015

# Combining Dynamic and Static Analysis for Malware Detection

Anusha Damodaran
*San Jose State University*

# Combining Dynamic and Static Analysis for Malware Detection

A Project

Presented to

The Faculty of the Department of Computer Science

San Jose State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Anusha Damodaran

May 2015

The Designated Project Committee Approves the Project Titled

Combining Dynamic and Static Analysis for Malware Detection

by

Anusha Damodaran

APPROVED FOR THE DEPARTMENTS OF COMPUTER SCIENCE

SAN JOSE STATE UNIVERSITY

May 2015

| | |
|---|---|
| Dr. Mark Stamp | Department of Computer Science |
| Dr. Thomas Austin | Department of Computer Science |
| Dr. Sami Khuri | Department of Computer Science |

## ABSTRACT

### Combining Dynamic and Static Analysis for Malware Detection

### by Anusha Damodaran

Well-designed malware can evade static detection techniques, such as signature scanning. Dynamic analysis strips away one layer of obfuscation and hence such an approach can potentially provide more accurate detection results. However, dynamic analysis is generally more costly than static analysis. In this research, we analyze the effectiveness of using dynamic analysis to enhance the training phase, while using only static techniques in the detection phase. Relative to a fully static approach, the additional overhead is minimal, since training is essentially one-time work.

## ACKNOWLEDGMENTS

I would like to express my gratitude to my graduate advisor Dr. Mark Stamp for his continuous guidance and support throughout this project. I would like to thank him especially for pushing me and believing in me.

I would like to thank Dr. Sami Khuri and Dr. Thomas Austin for their encouragement, support and time.

I would also like to thank Fabio for his help and support.

And, I would like to thank each and everyone in my family who have been kind and patient with me throughout my Masters.

Last but not the least, I would like to thank my dear husband for supporting and loving me no matter what.

## TABLE OF CONTENTS

**CHAPTER**

## LIST OF TABLES

## LIST OF FIGURES

# CHAPTER 1

## Introduction

"Malware is any kind of software intentionally developed for malicious purposes without user's awareness" [17]. Malware continues to be widespread despite the use of anti-virus software. There are many kinds of malware such as virus, botnet, worm, trojan, spyware etc., which is discussed in Chapter 2 in detail.

As per the 2015 Symantec Internet Security Report [45] which was released in April, more than 317 million new pieces of malware were created in the year 2014 alone, 26 percent more than the malware variants in 2013. Up to 28 percent of these were virtual machine aware. Out of which some like W32.Crisis not only are virtual machine aware, but also checks for other virtual machine images and infects them. Also, in 2014, the number of malicious spam emails surged from a mere seven percent in October to 41 percent in November. Most of these emails used Downloader.Upatre and Downloader.Ponik which are well known trojans that download other malware like Trojan.Zbot into compromised systems. Also, it is found that in virtualized environments, malware instead of quitting, started to exhibit behavior similar to benign software to avoid detection.

Due to the increasing number of new and unknown malware every year, analysis and detection of malware has been the prime research area these days. Various approaches are used for both analysis and detection of malware. The most commonly used method for the detection of malware is signature based detection. Anti-virus software widely use signature based detection methods which rely on exact signature matching. The main disadvantage of signature based detection is that it is only useful

for an already known malware attack. It is ineffective while detecting new malware or even variants of existing malware whose signatures have not been generated yet [17, 18].

A large number of malware use simple techniques such as code obfuscation, encryption to evade signature based detection. Due to the inefficiency of signature based detection for a new malware outbreak, malware researchers started focusing on using behavior based detection methods for the unknown malware attack.

Behavior based detection methods identifies the actions performed by the malware instead of syntactic markers like signatures [25]. Hence behavior based detection methods have large processing overheads than signature based detection methods. A few behavior based malware detection techniques have already been proposed in prior literature. A majority of these proposed techniques rely on extracting unique patterns from the instruction, system or API call graphs. The detection is typically done by graph matching algorithms, similarity based algorithms or sub-graph based features [28]. In addition to the above detection techniques, there are several other detection techniques which are discussed in Chapter 2.

These detection techniques primarily make use of analysis techniques to analyze a given malware and understand the intention of a malware [16]. There are three different approaches for malware analysis such as static, dynamic and hybrid analysis. Static analysis is analyzing a software without executing the software [29]. Static analysis usually involves disassembling a software and analyzing the opcode sequences, the control flow graphs. Other techniques involve analyzing the file characteristics, the byte codes from the executable file. Whereas, Dynamic analysis is analyzing the tasks performed by a program while it is being executed [16] in a virtualized environment. Dynamic analysis strips away one layer of obfuscation and hence such an approach can

potentially provide more accurate detection results. There are several techniques and approaches followed to perform dynamic analysis which will be discussed in Chapter 2. On top of these two techniques, there is one more analysis technique which is the hybrid analysis of malware which combines the advantages of both static and dynamic analysis [36]. In hybrid analysis, static analysis is done before execution of a program and then dynamic analysis is done selectively based on the information obtained from static analysis. Refer to Chapter 2 for more elaborate overview on analysis techniques.

The main concern as per Symantec is that, the attackers have been constantly upgrading their stealth techniques while security companies are still fighting old tactics [45]. There is a need to come up with more accurate detection techniques without compromising the efficiency. In this paper, we analyze the effectiveness of using dynamic analysis to enhance the training phase, while using only static techniques in the detection phase. Relative to a fully static approach, the additional overhead is minimal, since training is essentially one-time work. Also, in this approach scoring applies only the static approach which is faster. We use statistical based detection approach which has been used in the past and has been proven to give better results than commonly used detection approaches.

This paper is organized as follows. In Chapter 2, we discuss the various types of malware, provide background information on malware detection techniques and malware analysis techniques and discuss about the various tools used for malware analysis. In Chapter 3, we provide the implementation details of the project. In Chapter 4, we discuss the details of the experiments and analyze the results obtained. In Chapter 5, we conclude by giving the summary of results and discuss the possible future work.

# CHAPTER 2

# Background

Malware or Malicious Software as described in Chapter 1 either stops normal execution of benign programs or they start executing along with benign code to do malicious activities.

In this chapter, we describe some of the commonly known types of malware in the first section. In the second section, we discuss some of the well known malware detection techniques and elaborate on Hidden Markov Models. In the third section, we discuss the different malware analysis methods and elaborate previous work on malware detection. In the final section, we discuss the different malware analysis tools that are being used.

## 2.1 Types of Malware

There are several types of malware which can be differentiated by their characteristics such as the execution pattern, their internal structure and their method of propagation. They are as follows.

### 2.1.1 Virus

A virus is a type of malware that self-replicates into other existing executable programs. The infected code follows the same pattern to infect other existing code. Viruses spread within a computer or can travel to other computers via external media such as USB, CD/DVD, floppy disk, etc., [6]. Viruses cannot propagate through networks. A separate term 'worm' is used for malware that propagate through networks. Viruses can also be classified with respect to their concealment strategy as encrypted,

polymorphic and metamorphic malware.

#### 2.1.1.1 Encrypted Virus

Encrypted virus is a type of malware where, the entire body of malware is encrypted by using any of the known encryption techniques. The Code to decrypt the above created encrypted malware is also attached to its body. Signature based detection method fails to detect encrypted malware since the signature cannot be extracted from the malware. The malware body is decrypted only at the time of execution. Although, signature for the decryption code which is present in the executable itself can be used to detect such malware.

#### 2.1.1.2 Polymorphic Virus

In polymorphic virus, the code to decrypt malware is morphed after every infection. Morphing is done in such a way that the internal structure of code changes completely without any change in the functionality [34]. This overcomes the disadvantage of encrypted malware and signature based detection becomes quite impossible in this case.

#### 2.1.1.3 Metamorphic Virus

Metamorphic virus is an advanced version of polymorphic virus. In this malware, internal structure of the malware changes after every execution. But, the overall functionality remains the same [27]. Various approaches are used to morph the internal structure of the malware.

**Subroutines Permutation** In this method, malware writers rearrange the subroutine definitions within the code, while maintaining the ordering of subroutine

calls. This generates code that are functionally equivalent but structurally different.

**Instruction Reordering** Here, instructions are rearranged in a program as long as they satisfy the inter-instruction dependencies. That is, the instructions that are independent of previous instructions can be reordered to generate morphed copies [6, 34].

**Instruction Equivalence** Here, an instruction or a sequence of instructions are replaced by its functionally equivalent instruction to generate new variants of the same code [49].

**Register Renaming** This is the most simplest code obfuscation technique where we can simply rename all registers that are used and generate morphed copies [6].

**Inserting Junk Code** In this method, do-nothing instructions are inserted in software which will not affect the execution of program [13]. These do-nothing instructions are junk code or dead code that are added to facilitate code obfuscation.

Some of the other techniques include subroutine interleaving where subroutines are either merged or split to generate morphed copies or making spaghetti code where a lot of jump and go-to instructions are used. All of these techniques combined can be used successfully to generate quite a large number of morphed code copies [6].

### 2.1.2 Worm

A worm is similar to a virus in many ways. Worms also self-replicate themselves. The two distinct differences between worms and viruses are that worms are

stand-alone. They do not need existing executable code to infect. Also, worms can propagate through networks unlike viruses which need existing programs to help them propagate. A well known worm in the 80s is the Morris worm that crippled the internet for a certain duration of time [6].

### 2.1.3 Trojan

A Trojan is a type of malware which camouflages itself as a benign program but secretly performs hidden malicious activity. A classic example of a Trojan would be the password login program which prompts the username and password for an application and keylogs the password in the background. Then, it would flash an 'invalid password' message and then display the actual username, password prompt page [6]. Another example for a Trojan would be fake anti-virus programs that claim to remove viruses when in fact it performs malicious activity in the background. Some examples of Trojan viruses that are used in this research are Trojan.Zbot [47], Trojan.ZeroAccess [48], Trojan.Cleaman [46] etc.

### 2.1.4 Backdoor

A Backdoor is a mechanism which bypasses a normal security check [6]. One example of Backdoor that we use in this research is Harebot. Harebot.Backdoor allows to gain remote access to the infected system. It attempts to perform stealth activities to lock resources and affects the productivity of the system. It cannot propagate automatically on its own [24].

## 2.2 Detection Techniques

There are several detection techniques commonly used to detect a malware. The various characteristics of malware are extracted and used in detection techniques such as statistical analysis on opcode sequences, pattern mining or similarity based techniques etc.,. Some of the popular techniques are discussed as follows.

### 2.2.1 Signature Based Detection

Signature based detection is the commonly used malware detection technique in anti-virus software. A signature is a sequence of bytes that is unique to a particular malware which is constructed by security experts after analyzing the malware. Anti-virus software maintain a large repository of signatures of known malware [6]. These repositories are updated frequently to keep it up-to date. To detect whether an executable is benign or malware, the executable is matched with the signatures present in the database using signature matching algorithms like Aho-Corasick algorithm, Veldman algorithm, Wu-Manber algorithm etc.,. Signature based detection scheme is simple and fast [44]. The drawback of this technique is that it requires up-to date signature database. It cannot detect new variants of malware whose signatures are not present in the repository. Simple obfuscation techniques like metamorphism, polymorphism can be used to evade signature based detection [32].

### 2.2.2 Behavior Based Detection

Behavior based detection mainly focuses on the actions performed by the malware during execution and understanding the intent of the malware using various techniques. In behavior based detection, the behavior of the malware and benign files are studied during the training(learning phase) and then during the monitoring

phase, the executable is either classified as malware or benign. A slight variation gives rise to anomaly based detection wherein the behavior of all files are studied in the training phase, and during the monitoring phase any file which is deviated from normal behavior is classified as infected [25].

### 2.2.3 Statistical Based Detection

Statistical based detection has been very successful in recent times for malware detection. Hidden Markov Model (HMM) is one such statistical analysis method which is used to detect malware [43]. In this Project, we use HMM for malware detection. Hence HMM is explained in detail in upcoming sections.

#### 2.2.3.1 Hidden Markov Model

Hidden Markov Models are generally used for statistical based pattern analysis. They are used in most of the speech recognition systems and in numerous applications in computational molecular biology, pattern recognition, artificial intelligence and malware detection [22]. In the following sections, we will give brief introduction on HMM and its usage in malware detection. A Hidden Markov Model is a machine learning model to represent probability distributions over sequence of observations [22]. The Hidden Markov Model has two significant properties from which it attained its name. The first property is that it assumes that the observation at time t was generated by some process whose state $S_t$ is hidden from the observer. Second, it satisfies the markov property i.e., the current state $S_t$ is dependent only on $S_{t-1}$ and is independent of all states prior to t-1. A generic Hidden Markov Model is given in Figure 1 [43].

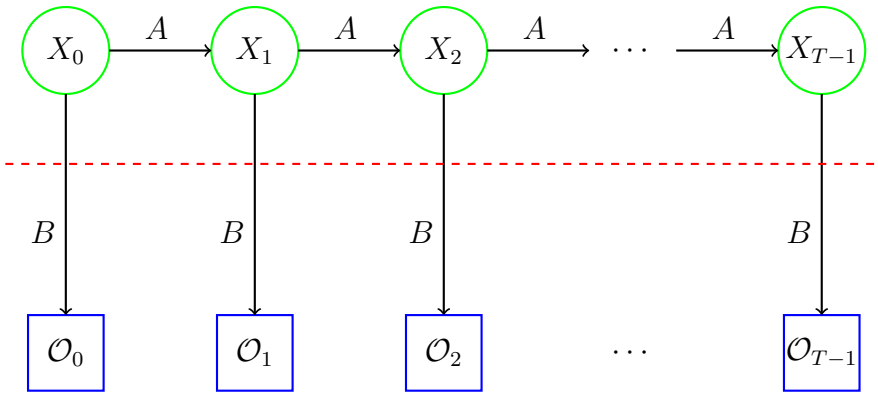


Figure 1: Hidden Markov Model

The notation used in HMM is as follows [43].

$$
\begin{array}{lcl}
T & = & \text{length of the observation sequence} \\
N & = & \text{number of states in the model} \\
M & = & \text{number of observation symbols} \\
Q & = & \{q_0, q_1, \ldots, q_{N-1}\} = \text{distinct states of the Markov process} \\
V & = & \{0, 1, \ldots, M-1\} = \text{set of possible observations} \\
A & = & \text{state transition probabilities} \\
B & = & \text{observation probability matrix} \\
\pi & = & \text{initial state distribution} \\
\mathcal{O} & = & (\mathcal{O}_0, \mathcal{O}_1, \ldots, \mathcal{O}_{T-1}) = \text{observation sequence.}
\end{array}
$$

Every state in an Hidden Markov Model represents a probability distribution of sequence of observations. The transition between states have fixed probabilities [43]. We train HMM using observation sequences that represent a particular dataset. We can then match an observation sequence against the model that was trained using HMM to determine it's score which is the probability of the occurrence of the observation in a given model. If the probability is high, then the scored observation sequence is very similar to the trained model. If the probability is less, then the observation sequence is very different to the trained model. A Hidden Markov Model is defined by the matrices $\pi$, $A$ and $B$. It is denoted by $\lambda = (A, B, \pi)$ [3, 43]. The three problems that can be solved using an HMM are [3, 43],

**Problem 1:** Given a HMM $\lambda = (A, B, \pi)$ and an observation sequence $\mathcal{O}$, determine $P(\mathcal{O}|\lambda)$. This is to determine the likelihood of the sequence given the model.

**Problem 2:** Given a HMM $\lambda = (A, B, \pi)$ and an observation sequence $\mathcal{O}$, determine the optimal state sequence for the markov process. Here, we uncover the hidden part of the HMM.

**Problem 3:** Given an observation sequence $\mathcal{O}$, $N$ and $M$, determine the model $\lambda = (A, B, \pi)$ that maximizes the probability of $\mathcal{O}$. Here, we determine a model that best fits the observation data.

In this project, we use Problem 1 and Problem 3. Problem 3 is used while training the observation sequences obtained from malware data to determine a model. Problem 1 is used to determine the log likelihood of the observation sequence obtained from either the malware or benign file which is used for testing. The score obtained from Problem 3 is used to determine if the observation sequence is similar to the trained model or not. If there is a significant difference in scores obtained from malware and benign files, then a threshold could be determined to classify the file either as malware or benign.

To solve the above mentioned problems, we have a few well known algorithms [43] such as,

- Forward Algorithm
- Backward Algorithm
- Baum-Welch Algorithm

An overview of the above algorithms as seen in [43] in given below. The forward algorithm, also known as $\alpha$ pass, can be used to solve Problem 1. It is used to determine $P(\mathcal{O}|\lambda)$. The algorithm is stated as follows [43].

For $t = 0, 1, \ldots, T-1$ and $i = 0, 1, \ldots, N-1$

$$\alpha_t(i) = P(\mathcal{O}_0, \mathcal{O}_1, \ldots, \mathcal{O}_t, x_t = q_i|\lambda)$$

Here, $\alpha_t(i)$ is the probability of the observation sequence up to time $t$. The probability of the complete sequence is then calculated as follows.

For $i = 0, 1, \ldots, N-1$, let

$$\alpha_0(i) = \pi_i b_i(\mathcal{O}_0)$$

For $t = 0, 1, \ldots, T-1$ and $i = 0, 1, \ldots, N-1$, calculate

$$\alpha_t(i) = \left[\sum_{j=0}^{N-1} \alpha_{t-1}(j) a_{ji}\right] b_i(\mathcal{O}_t)$$

Then the probability is given by,

$$P(\mathcal{O}|\lambda) = \sum_{i=0}^{N-1} \alpha_{T-1}(i)$$

The backward algorithm, or $\beta$ pass is used to find the most likely state sequence. The algorithm is as follows [43].

For $t = 0, 1, 2, \ldots, T-1$ and $i = 0, 1, 2, \ldots, N-1$

$$\beta_t(i) = P(\mathcal{O}_{t+1}, \mathcal{O}_{t+2}, \ldots, \mathcal{O}_{T-1}, x_t = q_i, \lambda)$$

Then, $\beta_t(i)$ is calculated as follows.

For $i = 0, 1, 2, \ldots, N-1$, let

$$\beta_{T-1}(i) = 1$$

For $t = T-2, T-3, \ldots, 0$ and $i = 0, 1, 2, \ldots, N-1$, calculate

$$\beta_t(i) = \sum_{j=0}^{N-1} a_{ij} b_j(\mathcal{O}_{t+1}) \beta_{t+1}(j)$$

For $t = 0, 1, 2, \ldots, T-1$ and $i = 0, 1, 2, \ldots, N-1$, $\gamma_t(i)$ is defined as

$$\gamma_t(i) = P(x_t = q_i | \mathcal{O}, \lambda)$$

Then, $\gamma_t(i)$ is given as,

$$\gamma_t(i) = \alpha_t(i)\beta_t(i)/P(\mathcal{O}|\lambda)$$

In all likelihood, the state at time $t$ is $q_i$ where $\gamma_t(i)$ is maximum.

To solve Problem 3, we have to adjust the parameters of the model $A$, $B$ and $\pi$ to best fit the observations. N, the number of states and M, the number of observation symbols are fixed, but The model parameters $A$, $B$ and $\pi$ has to be determined in every iteration of the hill-climb with the row stochastic condition. Baum-Welch algorithm helps in effectively re-estimating the parameters [43].

For $t = 0, 1, 2, \ldots, T-1$ and $i, j \in \{0, 1, 2, \ldots, N-1\}$, the di-gammas are defined as,

$$\gamma_t(i, j) = P(x_t = q_i, x_{t+1} = q_j | \mathcal{O}, \lambda).$$

The di-gammas is the probability of transitioning from state $q_i$ at time $t$ to state $q_j$ at time $t+1$. It is written as,

$$\gamma_t(i, j) = \alpha_t(i) a_{ij} b_j(\mathcal{O}_{t+1}) \beta_{t+1}(j)/P(\mathcal{O}|\lambda).$$

The relation between di-gammas and $\gamma_t(i)$ is given by,

$$\gamma_t(i) = \sum_{j=0}^{N-1} \gamma_t(i,j)$$

Problem 3 can then be solved by the following steps [43].

- Initialize $\lambda = (A, B, \pi)$. We choose random values instead of uniform values to escape the local maximum during the hill climb process. The values are chosen such that $\pi_i \approx 1/N$, $A_{ij} \approx 1/N$ and $B_{jk} \approx 1/M$.
- Then, calculate $\alpha_t(i)$, $\beta_t(i)$, $\gamma_t(i)$ and $\gamma_t(i,j)$ as above.
- Re-estimate the parameters $A$, $B$ and $\pi$ as follows.

  For $i = 0, 1, \ldots, N-1$, let

  $$\pi = \gamma_0(i)$$

  For $i = 0, 1, \ldots, N-1$ and $j = 0, 1, \ldots, N-1$, calculate

  $$a_{i,j} = \frac{\sum_{t=0}^{T-2} \gamma_t(i,j)}{\sum_{t=0}^{T-2} \gamma_t(i)}$$

  For $j = 0, 1, \ldots, N-1$ and $k = 0, 1, \ldots, M-1$, calculate

  $$b_j(k) = \frac{\sum_{t \in 0,1,\ldots,T-1, \mathcal{O}_t = k} \gamma_t(j)}{\sum_{t=0}^{T-2} \gamma_t(j)}$$

- If probability $P(\mathcal{O}|\lambda)$ increases, then repeat from step 2.

We can stop the process when $P(\mathcal{O}|\lambda)$ does not increase by some predetermined threshold or when it reaches the maximum number of iterations [43].

#### 2.2.3.2 Malware Detection Using Hidden Markov Model

Now, we will look at the basic steps followed for performing malware detection using Hidden Markov Model. First, we select the observation data that the model should be trained for. In this case, the observation sequences represent a set of malware data. The observation sequences can be API calls sequence or opcode sequences. A model is trained with the above observation sequences. This can be related with Problem 3. After convergence, we get an accurate model that best fits the observation sequences.

Next, we score a set of malware and benign files against the trained model. If the scores are separated with the help of a predetermined threshold, the scores above the threshold can be said to be files from the malware family and ones that are below the threshold can be said to be files from the benign family. The above method needs a predetermined threshold to classify between malware and benign files. The accuracy of the detection method is then calculated depending on the number of correct classifications. The above approach in calculating accuracy in practice is not a good idea. A better approach would be to compute AUC to determine the accuracy. An AUC score close to 1.0 is the best score. More details on this can be found at [52]. In Chapter 3, we shall see how HMM has been used for malware detection in this project in detail.

## 2.3 Analysis Techniques

The three main malware analysis techniques are static, dynamic and hybrid analysis. Each analysis method has its own advantages and disadvantages which has been discussed in this section.

### 2.3.1 Static Analysis

Static analysis is the analysis of a software performed without actually executing the program [16]. Various approaches are followed to perform static analysis. Some of them are based on the characteristics of the binary file itself, by extracting byte code sequences from the binary, by extracting opcode sequences after disassembling the binary file, by extracting control flow graph from the assembly file, by extracting API calls from the binary and so on. Each represents a feature set and any one or a combination is used for malware detection.

In [10], the authors presented a technique to detect malicious behavior in executables using static analysis. They generate an annotated CFG of an executable and make use of an automaton that represents malicious code to detect malware. Their approach focuses on detecting obfuscation patterns in malware and succeed at that by showing good accuracy.

In recent years, detection of metamorphic malware has been very much effective due to the application of Markov models to malware detection as discussed before. Profile Hidden Markov Models(PHMM) is known for their success in determining relations between DNA and protein sequences. When applied for malware detection it has been found that Profile Hidden Markov Models can effectively detect metamorphic malware [43]. HMM has also been successful in this regard. In [3], HMM was also used for malware classification.

In [14], the malware detection is done based on function call graph analysis. In [41], opcode-based similarity measure was developed similar to simple substitution cryptanalysis method. The paper shows excellent results for metamorphic malware detection.

In [40], the corresponding advantages of API call sequences and assembly code are combined and a similarity based matrix is produced that determines whether a portion of code has traces of a particular malware. Two detection methods that have been discussed in this paper are SAVE(Static Analyser for Vicious Executables) that focuses on assembly calls and MEDiC(Malware Examiner using Disassembled Code) that uses API calls for analysis.

### 2.3.2 Dynamic Analysis

Dynamic Analysis is the analysis of a software performed while executing the program [16]. Some of the information that can be obtained by dynamic analysis are API calls, system calls, instruction traces, taint analysis, registry changes, memory writes etc.,. Some of the malware detection techniques using dynamic analysis which has been researched in the past are as follows.

In [26], the authors presented a malware detection technique using dynamic analysis where fine-grained models are built to capture the behavior of malware using system calls information and use a scanner to match the activity of any unknown program against these models to classify them as either benign or malware. The behavior models are represented in the form of graphs. The vertices denote the system calls and the edges denote the dependency between the calls where the input of one system call(vertex) depends on the output of another system call(vertex).

In [1], the authors propose a run-time monitoring based malware detection tool that extracts statistical features from malware using spatio-temporal information from API call logs. The spatial information is the arguments and return values of the API calls and temporal information is the sequence of the API calls. This information is used to build formal models that are fed to standard machine learning algorithms

for malware detection. In [17], a set of program API calls is extracted and combined with the control flow graphs to obtain a new model API-CFG where API calls form the edges in the control flow graph. This API-CFG is trained by a learning model and used as a classifier during the testing stage for malware detection. In a slightly modified version, [19] they apply n-grams method to the API calls extracted and used the above as a feature set for malware classification. In [23], dynamic malware detection is done using registers values set analysis.

Some of the works focus on kernel memory mapping to develop a malware behavior monitor while analyzing kernel execution traces [35]. The authors in [30] propose a faster and effective method for malware classification using detection of maximum common subgraph algorithms.

In [53], the API sequences are extracted from a PE file and OOA based mining is done for malware classification. In [33], malware is analyzed by abstracting the frequent itemsets in API call sequences. In [31], a kernel object behavioral graph is created and graph isomorphism techniques and weighted common graph technique is used to calculate the hotpath for each malware family. And the unknown malware is then classified into whichever malware family has similar hotpaths.

In [12], dynamic instruction sequences are logged and are converted to abstract assembly blocks. Data mining algorithms are used to build a classification model using feature vectors extracted from the above data. For detection, the same method is used and scored against the classification model.

In [2], the authors propose a malware detection technique that uses instruction trace logs of executables collected dynamically. These traces are then constructed as graphs. The instructions are considered as nodes and the data from the instruction

trace is used to calculate the transition probabilities. Then a similarity matrix is generated between the constructed graphs using different graph kernels. Finally, the constructed similarity matrix is fed to a SVM for classification.

From all the research given above, it is evident that dynamic analysis can be viewed as a good source of information on malware behavior. Even though, they incur an execution overhead, an accurate model can be obtained from dynamic analysis. Also, the research has shown good results by using API call sequences and Opcode sequences to give almost a perfect description of the behavior of a malware.

### 2.3.3 Hybrid Analysis

Hybrid analysis technique is the combination of static and dynamic analysis techniques. Dynamic analysis can be tedious due to multi-path execution. Static analysis can be used to selectively choose the path of execution for dynamic analysis and hence can increase accuracy and efficiency [36].

In [9], the authors propose an innovative framework for classification of malware using both static and dynamic analysis. They define features or characteristics of malware as Mal-DNA(Malware DNA). Mal-DNA combines static, dynamic and hybrid characteristics. Their framework contains three components:

1. Static Analyzer using various techniques - START extracts the static characteristics of malware.

2. Debugging-based behavior monitor and analyzer - DeBON extracts hybrid and dynamic characteristics of them.

3. CLassifier using Mal-DNA - CLAM classifies malware based on extracted Mal-DNA from 1 and 2 using machine learning algorithms.

#### 2.3.3.1 HDM-Analyser

In [18], the authors develop HDM Analyser where they use both static analysis and dynamic analysis techniques in the training phase and perform only static analysis in the testing phase. The authors mainly focus on taking the advantage of accuracy of the dynamic analysis and the advantage of time complexity from static analysis. By combining the static and dynamic analysis techniques, they have showed that HDM-Analyser has better overall accuracy and time complexity than static and dynamic analysis methods [18]. The authors extracted a sequence of API calls for dynamic analysis data which is one of the most effective feature which can describe the behavior of a program perfectly.

Instead of using both dynamic and static analysis as used in [18], in this project only dynamic analysis techniques has been used to train the learning model. We then score the observation data obtained from static analysis. Since the results obtained from malware detection using HMM has been consistently good, HMM is used to test the technique here.

## 2.4 Tools used for Dynamic and Static Analysis

Elaborate details and links to all tools are given in the RCE Tool Library [11]. The library consists of disassemblers, debuggers, obfuscation tools, decompilers, etc.,. Some of the tools that are used to perform both dynamic and static analysis on Win32 files are given below.

**IDA Pro** is one of the best disassembler that generates near to accurate assembly language source code from executable code. It can also be used as a debugger. It supports various file formats and also uses IDC scripts and IDA Python scripts

to help in debugging. The features 'function tracing' and 'instruction tracing' enables to log all instruction executions and function calls. In this project, we use IDA Pro to generate .asm files from which opcodes and windows API calls can be extracted. Also, we use IDA Pro to collect the instruction trace logs of executables.

**OllyDbg** is a 32-bit disassembler and debugger, which is second best to IDA Pro. OllyDbg has limited features compared to IDA Pro.

### 2.4.1 Dynamic Analysis Tools

Some of the dynamic analysis tools that are free of cost or open source tools that can be used to extract API calls or instruction execution logs are as follows.

**Ether malware analysis tool** is an open source tool that has been developed via hardware virtualization extensions, and resides completely outside of the target OS. This enables no in-guest software components detection and hence no problems of incomplete or inaccurate emulation. Most of the recent viruses can detect a debugger or a virtual environment during execution. Ether malware analysis tool successfully overcomes the above problem and hence been a great tool for research purposes [15].

**API Logger** is a simple tool that logs all API calls that satisfy the inclusion and exclusion list. The inclusion list specifies which libraries need to be included and exclusion list specifies which libraries or functions can be ignored [20].

**WinAPIOverride** is an advanced API monitoring software. It tries to fill the gap between classic API monitoring software and debuggers. It can be very useful to manually extract API calls by deciding the flow of the program during execution.

**API Monitor** is a software that also helps in monitoring and controlling API calls made by applications of processes. This tool needs to be run inside a virtual machine to analyze a malware and cannot be run in a sandboxed environment [4].

**Buster Sandbox Analyzer** is a dynamic analysis tool that has been designed to determine if processes exhibit malicious activities. In addition to analyzing the behavior of the processes, it keeps track of the changes made to the system such as registry changes, file-system changes and port changes [8]. The tool runs inside a Sandbox which protects the system from getting infected while executing the malware. The sandbox that is being used by Buster Sandbox Analyzer is Sandboxie [38]. In this project, we use BSA to generate API calls for win32 executables.

There are several other frameworks such as Detours, DynInst for Windows API tracing. There are also several other tools that uses various approaches to trace Windows API functions in-guest, some of which are mentioned above. And several tools similar to Buster Sandbox Analyzer that are used to trace in sandboxed-environments are CWSandbox and Norman Sandbox [15].

In the next chapter, we will see how these tools are used to collect data and other implementation details of the project.

# CHAPTER 3

# Implementation

In this chapter, we first give an overview of the malware dataset used in this project. Then we elaborate on extraction of API call sequences and opcode sequences to be used in the project. In the final section of this chapter, we discuss the implementation details of the project.

## 3.1 Chosen Dataset

The following seven different malware families were considered as malware dataset for this project.

**Trojan.Zbot** also known as Zeus, is a trojan horse which compromises the system by downloading configuration files or updates and attempts to steal confidential information. It is a stealth virus which hides in the file system within moments of execution. The virus vanishes from processes list and the most IDA Pro could track the virus is for the initial 5 - 10 minutes of execution [47].

**Trojan.ZeroAccess** is also a trojan horse which makes use of an advanced rootkit to hide itself. It is capable of creating a new hidden file system, create a backdoor in the compromised system and download new malware [48].

**Win32/Winwebsec** is a rogue security software, that pretends to be an anti-virus software. When infected, it displays fake messages showing malicious activity and tricks the customers into paying money for the software to clean up the system [51].

**Harebot.M** is a backdoor which attempts to gain remote access of the infected system and affects the productivity of the system and network. It attempts to connect to other remote sites and awaits orders from these remote sites [24].

**S.M.A.R.T HDD** is similar to Winwebsec, also a fake rogue security software that offers to do registry cleaning and malware removal. It starts a fake scanning process and shows fake messages of detected trojan, viruses. And it does not let the user to delete the detected viruses unless the user purchases the license of the software [42].

**Security Shield** is similar to Winwebsec and Smarthdd. It is a Trojan where it pretends to be a fake anti-virus software. It causes annoyance to users by repeated fake virus detection messages and coerces the users into purchasing the software [39].

**Trojan.Cleaman** is a trojan horse that arrives as an email attachment. It may use an invalid digital certificate to pose as a legitimate file. It also infects itself into all running processes. It checks network traffic and redirects the browser to a remote IP address whenever the user requests a particular URL [46].

For the benign dataset, the dataset was collected from Windows System 32 files. These files are present in every windows system and are used often. Some of the files that are used in the benign set are notepad.exe, calc.exe, sort.exe, solitaire.exe etc.,. Table 1 shows the number of files used from each family from the malware dataset. All experiments in 4 were performed with the dataset present in the Table 1.

Table 1: Malware Dataset

| Family | No. of Files |
|---|---|
| Zbot | 200 |
| ZeroAccess | 200 |
| Winwebsec | 200 |
| Harebot | 49 |
| Smarthdd | 53 |
| Security Shield | 54 |
| Cleaman | 32 |

### 3.2 Observation Data : API call sequences

Microsoft Windows Operating System provides a variety of API calls that facilitate in requesting services from the kernel of the operating system [1]. Each executable's execution flow is analogous to its sequence of API calls. In the past, information obtained from API calls has been used to study the behavior of malware.

#### 3.2.1 Data Extraction

Each API call has a distinct name, a set of arguments and return value. We focus only on API call names to construct the call sequence. The calls to Windows Application Programming Interface (API) can be extracted from all executables dynamically or statically. The extraction of an API call sequence for a particular malware has been explained in the following sections. A sample partial extracted API call sequence for a malware executable is shown in Listing 3.1.

Listing 3.1: Sample API call sequence

```
...OpenMutex CreateFile OpenProcessToken AdjustTokenPrivileges
   SetNamedSecurityInfo LoadLibrary CreateFile GetComputerName
   QueryProcessInformation VirtualAllocEx DeleteFile...
```

#### 3.2.1.1 Dynamic Analysis

There are various tools to generate API call logs from an executable dynamically as mentioned in Chapter 2, Section 2.4. In this Project, we use Buster Sandbox Analyser(BSA) tool [8] to generate API calls by dynamic analysis. Buster Sandbox Analyser uses a sandboxed environment to execute the malware and logs all API calls for a certain duration of time. From the logged API reports, API calls are extracted and merged together to form an API call sequence for each executable which is used as observation data for training the HMM model. Listing 3.2 shows a sample API log extracted via BSA.

Listing 3.2: Sample API log from BSA

```
...
QuerySystemInformation(SystemBasicInformation)
QuerySystemInformation(SystemProcessorInformation)
QueryProcessInformation(00
   b6b7e3b923861ef8c257aa3803a239ce4d6154.exe,
   ProcessImageInformation)
VirtualAllocEx(00b6b7e3b923861ef8c257aa3803a239ce4d6154.exe,
   MEM_COMMIT MEM_RESERVE, PAGE_EXECUTE_READWRITE, RegionSize=0
   x00015500)
LoadLibrary(ole32.dll)
...
```

#### 3.2.1.2 Static Analysis

For the static analysis part, IDA Pro was used for extracting API calls. An API call set is usually obtained by Import Address Table(IAT) analysis which extracts API information from the table that stores this information. To obtain an API call sequence that is similar to the sequence obtained dynamically, we take a different approach by disassembling the file and extracting the API calls in the same order as it is present in the .asm file. The call instructions in a .asm file consist of subroutine calls, API calls, near/far calls and several others out of which API calls are alone

extracted which is constructed into a sequence. These extracted sequences obtained by static analysis are used to test against the trained HMM Model.

## 3.3 Observation Data : Opcode sequences

Similar to API call sequences, opcode sequences have also been used widely in the past for malware detection. Opcode sequences deals with the source code and when used with HMM has proven to be effective for detection of metamorphic malware.

### 3.3.1 Data Extraction

Opcodes or operational codes are the portion of the assembly language instruction which specifies the operation. The following section explains extraction of opcode sequences dynamically and statically. A sample opcode sequence that is extracted is shown in the listing 3.3

Listing 3.3: Sample opcode sequence

```
... push mov call mov push call add mov pop retn push mov call
   mov push call add mov pop ...
```

#### 3.3.1.1 Dynamic Analysis

For Dynamic Analysis, Ether malware analysis tool and IDA Pro were used for extraction of opcode sequences. Both tools generate an instruction trace log during the execution of an executable from which the opcode sequence is constructed. In Ether, a feature called instruction tracing was used to trace all instructions executed in a log for a certain period of time. To perform instruction tracing in Ether, we first execute the executable in the guest OS, and issue the command `ether dom-id instrtrace exe-file` in the host OS where `dom-id` is the guest domain where the

executable is running and `exe-file` is the process name that has to be traced. The ether controller smartly sets a trap flag after execution of each instruction, hides the debug exception evoked from the analysis target and logs the instruction which prevents the executable to realize the presence of a debugger. The result is an accurate log of executed instructions which is saved in the host OS. A sample section of the ether instruction trace log is shown in Listing 3.4.

Listing 3.4: Sample instruction trace log from Ether

```
...
40fb9f: jmp     0x0040FB58
40fb58: mov     [ebp-0x18], 0x00000000
40fb5f: lea     eax, [ebp-0x18]
40fb62: push    eax
40fb63: push    [ebp-0x14]
40fb66: push    edi
40fb67: call    0x00403DD8
...
```

Using IDA Pro, instruction trace logs are obtained for executables via the 'Tracing' feature. After disassembling the executable, a breakpoint is placed at the entry point of the program. Then, the instruction tracing feature is enabled. Several options such as 'trace over library functions' etc can be enabled/disabled in Tracing Options. The process then begins execution and logs each instruction that is being executed. Since it writes each instruction the normal execution time cannot be compared with the execution time in IDA Pro with 'Instruction Tracing' on.

A sample section of the IDA instruction trace log shown in Listing 3.5. Opcode sequences are generated from the instruction trace log by extracting the opcodes in the same ordering as it appears in an .asm file and merging them. These opcode sequences are used as observation data for training the HMM model.

Listing 3.5: Sample instruction trace log from IDA

```
...
00000AB4  .text:start           push ebp          ESP=12FF88
00000AB4  .text:start+1         mov  ebp, esp     EBP=12FF88
00000AB4  .text:start+3         call sub_4011B0   ESP=12FF84
00000AB4  .text:sub_4011B0      push ebp          ESP=12FF80
00000AB4  .text:sub_4011B0+1    mov  ebp, esp     EBP=12FF80
00000AB4  .text:sub_4011B0+3    sub  esp, 2CCh    ESP=12FCB4 AF
   =1 ZF=0
00000AB4  .text:sub_4011B0+9    push ebx          ESP=12FCB0
...
```

#### 3.3.1.2 Static Analysis

To extract opcodes by static analysis, the executables are disassembled and a seperate .asm file is generated for each executable. From the .asm files, opcodes are extracted by a simple pattern matching script excluding macros, assembler directives etc. The opcode sequences obtained by static analysis are used to score against the trained model.

### 3.4 Training the HMM and scoring

For both API call sequences and opcode sequences, three cases were implemented for comparison. The first is using dynamic analysis data alone for both training and testing. The second case is using dynamic analysis data for training and static analysis data for testing which is our proposed method. The third case is using static analysis data alone for both training and testing. We already know that dynamic analysis data is more accurate. We hope to see that when we train the model using dynamic data, and test using only static data, we can augment the accuracy advantage of dynamic case to the static case and hence hope to see that it is more accurate than the static case alone. If we succeed to bring results, then not only our proposed method will be more efficient than the dynamic case, but also more accurate than the static case. We used the five-fold cross validation method for training and scoring

the HMM. Each family is divided into five equal subsets. Four subsets are used for training and one subset is used for testing the HMM along with the benign set. This process is repeated five times till all five subsets are scored against the trained model. The dataset used is as shown in Table 1.

### 3.4.1 Case 1: Training - Dynamic data, Testing - Dynamic data

In this case, we train the HMM with either API call sequences or opcode sequences of a single family obtained by dynamic analysis. The trained model is tested against API call sequences or opcode sequences of the same family and benign set obtained by dynamic analysis. The observation sequences used are extracted as mentioned in Section 3.2.1.1 and Section 3.3.1.1.

### 3.4.2 Case 2: Training - Static data, Testing - Static data

In this case, we train the HMM with either API call sequences or opcode sequences of a single family obtained by static analysis. The trained model is tested against API call sequences or opcode sequences of the same family and benign set obtained by static analysis. The observation sequences used are extracted as mentioned in Section 3.2.1.2 and Section 3.3.1.2.

### 3.4.3 Case 3: Training - Dynamic data, Testing - Static data

In this case, we train the HMM with either API call sequences or opcode sequences of a single family obtained by dynamic analysis. The trained model is tested against API call sequences or opcode sequences of the same family and benign set obtained by static analysis. The observation sequences used for training are extracted as mentioned in Section 3.2.1.1 and Section 3.3.1.1. The observation sequences used

for testing are extracted as mentioned in Section 3.2.1.2 and Section 3.3.1.2. We mention this case as dynamic/static case or the mixed case in this project which refers to one and the same.

The results for the above three cases and the various experiments performed are elaborated in the next chapter.

# CHAPTER 4

# Experiments

In this chapter, we discuss about the experiments performed and discuss the results. Table 2 shows the specifications of the host and the guest that was used to perform dynamic analysis of malware executables using Buster Sandbox Analyzer and IDA Pro and Table 3 shows the specifications of the host and guest used along with Ether malware analysis tool.

Table 2: System Configuration

| **Host** | |
|---|---|
| Model | Dell Inspiron 15 |
| Processor | Intel Pentium(R) Dual-Core CPU T4300 @ 2.10GHz |
| RAM | 4.00 GB |
| System Type | 64-bit OS |
| Operating System | Windows 7 |
| **Guest** | |
| Virtualization Software | Oracle Virtual Box 4.3.16 |
| Base Memory | 512MB |
| System Type | 32-bit OS |
| Operating System | Windows XP |

The experiments were performed with API call sequences and opcode sequences as observation data for Hidden Markov Model. The effectiveness of each experiment is assessed with the help of ROC Curves and AUC.

## 4.1 ROC curves and AUC as a performance measure

ROC curves are an effective way to visualize the performance of a machine learning classifier by varying the decision threshold [7]. This helps in comparison of various

Table 3: System Configuration for Ether malware analysis tool

| **Host** | |
|---|---|
| Model | Dell XPS |
| Processor | Intel(R) Core(TM) i5-3230M CPU @ 2.60GHz |
| RAM | 6.00 GB |
| System Type | 64-bit OS |
| Operating System | Debian Lenny 5.0 Vanilla Install |
| **Guest** | |
| Virtualization Software | Xen Hypervisor 3.1.0 |
| Kernel | xen-linux-system-2.6.26-1-xen-amd64 |
| Base Memory | 512MB |
| System Type | 32-bit OS |
| Operating System | Windows XP |

techniques with a single identifier which is the Area Under the Curve(AUC) calculated from the ROC curve. ROC curves are plotted as two dimensional graphs with true positive rate(TPR) also known as sensitivity in Y-axis and false positive rate(FPR) also known as specificity in X-axis with varied thresholds. TPR is defined as the number of true positives over the actual number of positives. FPR is defined as the no. of false positives over the actual number of negatives. AUC is the area under the ROC curve which is plotted in a unit square. Since, AUC is just a portion of a unit square it can range from 0.5 to 1.0, 1.0 being the highest and 0.5 being the lowest. The performance of a classification is excellent if the AUC is close to 1.0 and considered a failure if the AUC is close to 0.5 [7, 21].

### 4.2 Discussion of results : Using API calls sequence

In this experiment, we train separate HMM models with API call sequences as observation data obtained from seven different families. The training and testing data is varied for all three cases as mentioned in Chapter 3, Section 3.4. Table 4 shows

the AUCs obtained for each of the cases.

Table 4: AUCs for API calls sequence

| Family | Dynamic | Static | Dynamic/Static |
|---|---|---|---|
| Security Shield | 0.9875 | 1.0000 | 0.9563 |
| Zbot | 0.9800 | 0.9936 | 0.9555 |
| Winwebsec | 0.9762 | 0.9647 | 0.7631 |
| Smarthdd | 0.9808 | 0.8225 | 0.7500 |
| ZeroAccess | 0.9968 | 0.9976 | 0.6526 |
| Harebot | 0.9867 | 0.8282 | 0.6321 |
| Cleaman | 1.0000 | 1.0000 | 0.5000 |

In the first column in Table 4, we have the dynamic/dynamic case where we train the HMM using dynamically obtained data and test with the same. As expected, the AUCs are closer to 1.0 for all families. In the second column, we have the AUCs obtained for the static/static case. The accuracy has exceeded slightly for some of the families and decreased for Smarthdd and Harebot families. Both of these cases although having good results, have disadvantages practically. The first case has execution overhead during the testing phase, while the second case can be easily defeated by simple code obfuscation techniques like adding dead code that is similar to benign files.

In the third column, we have the AUCs obtained by training HMM with dynamic data and testing the trained model with static data. We expect to augment the accuracy of the static case by having an accurate trained model with dynamic data. We see that for the families Zbot and Security Shield the performance is very good. The ROC Curve is shown in Figure 2.

Apart from Zbot and Security Shield, none of the other families have given good results. Also, none of them have been able to exceed the static analysis case which was

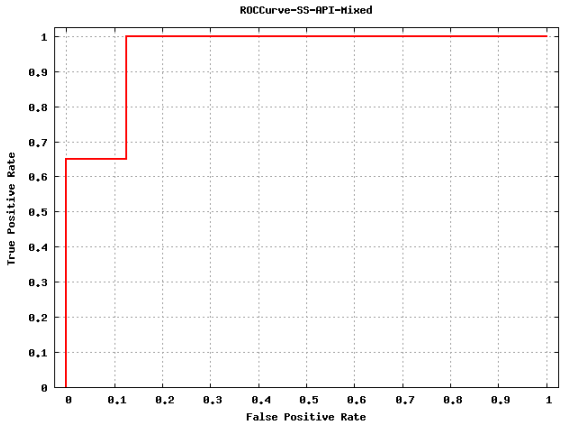


Figure 2: ROC for Security Shield-Dynamic/Static-API

the expected result. The reason being that, in each executable very less number(5 to 20) of API calls were obtained statically. When scored against a model trained with different(dynamic) data, lack of data for testing might have led to inconsistent results. Hence, we use opcode sequences in our next experiment instead of API call sequences. HMM using opcode sequences has been very effective for malware detection in the past.

### 4.3 Discussion of results : Using opcode sequence

In this experiment, we use opcode sequences for training and scoring the HMM for three cases similar to the previous section as mentioned in Chapter 3, Section 3.4. Initially for the first experiment, we use Ether malware analysis tool to obtain dynamic data and test using static data obtained from IDA Pro. Then in the second experiment, we use IDA Pro for both dynamic and static data. Our third experiment was based on the opcodes present only in dynamic data. We also experimented on data collected for a longer duration of time during dynamic analysis. Although not practical, we decided to perform the Static/Dynamic case too to obtain more information on the mixed case. All the results obtained from the above mentioned

experiments are discussed in the following sections.

### 4.3.1 Experiment 1: Ether/IDA Pro

Here, we use Ether malware analysis tool to generate opcode sequences from the instruction trace logs that are generated during dynamic analysis as explained in Chapter 3, Section 3.3.1.1. Ether malware analysis tool is an open source tool that has been developed via hardware virtualization extensions, and resides completely outside of the target OS. And hence, has the biggest advantage of having no problems with in-guest detection. As already mentioned in Chapter 1, around 28 percent of malware variants found in 2014 were virtual machine aware. They either quit abruptly as soon as they realize the VM environment, or perform tasks similar to benign files to avoid detection. Ether malware analysis tool smartly avoids detection and has total control on invocation of debug exceptions. Hence, dynamic analysis in Ether generates near to accurate logs with no incomplete or inaccurate emulation. We train the HMM using data obtained from Ether malware analysis tool. And then test the model with static data obtained from IDA Pro. The results obtained are shown in Table 5.

Table 5: AUCs for Ether/IDA opcode sequence

| Family | Dynamic | Static | Dynamic/Static |
|---|---|---|---|
| Zbot | 1.0000 | 0.6728 | 0.7767 |

The ROC curve for the mixed(dynamic/static) case corresponding to AUC 0.7767 is shown in Figure 3. From the results in Table 5, we see that the dynamic case does very well. And the mixed case does pretty well and better than the static case. Also, we learned that, since the dis-assembly by Ether and IDA are quite different, the discrepancies might be affecting the results. Hence, we tried the next experiment

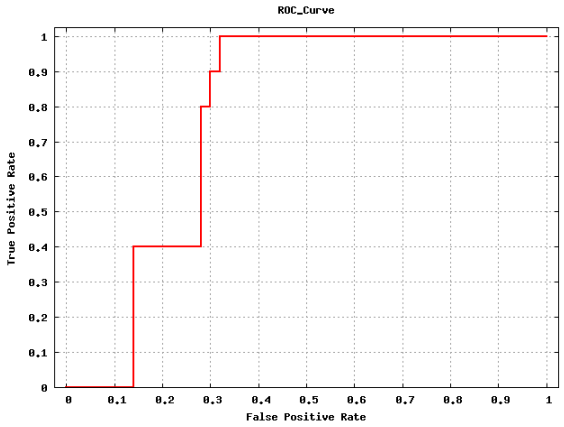


Figure 3: ROC for Zbot-Dynamic(Ether)/Static(IDA)-Opcodes

with both dynamic and static data from IDA Pro.

### 4.3.2 Experiment 2: IDA/IDA

Here, we train the HMM using dynamic data obtained from IDA Pro. And then test the model with static data obtained from IDA Pro. The instruction trace logs were generated as explained in Chapter 3, Section 3.3.1.1. We first test for a small subset of Zbot files. The results obtained are shown in Table 6.

Table 6: AUCs for IDA/IDA opcode sequence for a small subset

| Family | Dynamic | Static | Dynamic/Static |
|---|---|---|---|
| Zbot | 0.9650 | 0.5575 | 0.8525 |

From the above results we see that, the score for the mixed case(0.8525) has improved than the Ether/IDA experiment where the score was 0.7767. The ROC curve for the dynamic/static case is shown in Figure 4. Also, the mixed score is better than that of the static case. Hence, we try the same for all families for a larger dataset. The results are shown in Table 7.

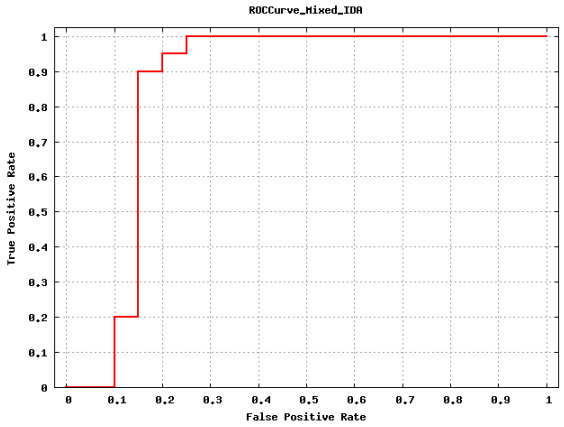


Figure 4: ROC for Zbot-Dynamic/Static-IDA-Opcodes

Table 7: AUCs for opcode sequence

| Family | Dynamic | Static | Dynamic/Static |
|---|---|---|---|
| Smarthdd | 0.9860 | 0.9952 | 1.0000 |
| Zbot | 0.9681 | 0.7755 | 0.6997 |
| Cleaman | 1.0000 | 0.8567 | 0.6889 |
| Winwebsec | 0.8268 | 0.6609 | 0.6064 |
| Harebot | 0.7210 | 0.5300 | 0.5961 |
| Security Shield | 0.9452 | 0.5028 | 0.5752 |
| ZeroAccess | 0.9840 | 0.7760 | 0.5600 |

From the results in Table 7, we see that we get a perfect score of 1.0 for Smarthdd family for the mixed case. The ROC Curve for the Smarthdd family is shown in Figure 5. But, the results for other families except Smarthdd family are not as expected for the mixed case. This was mostly due to the fact that we obtain only 50 to 80 distinct opcodes during dynamic analysis. Whereas, we obtain 230 to 280 distinct opcodes from the .asm files. This was in contrast with the number of distinct opcodes for Smarthdd family. Figure 6 shows the number of distinct opcodes obtained for each family in both static and dynamic case. The number of distinct dynamic opcodes for Smarthdd family was 34 and the number of distinct static opcodes was 81. The vast difference between the training set and testing set for other families

might have contributed to inconsistent results. To test this, the next experiment was performed by removing opcodes while scoring that is not present in the dynamic data used for training. that is, all opcodes not seen prior by the trained model was skipped while scoring.

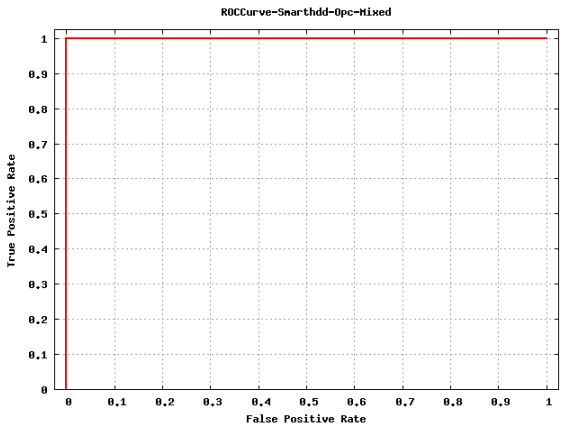


Figure 5: ROC for Smarthdd-Mixed-Opcodes

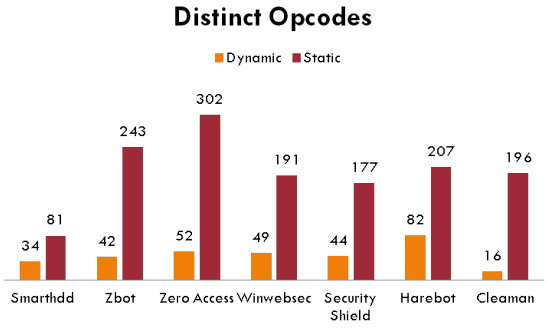


Figure 6: Number of Distinct Opcodes

### 4.3.3 Experiment 3: Skipping unseen opcodes

The next experiment that we conducted was to make the data obtained from static analysis to be similar to the training set. For example, for Zbot family, there are 42 distinct opcodes in the dynamic data and 243 distinct opcodes in static data. Hence, when the model is trained only for 42 observation symbols and tested with 243

symbols, it encounters a large number of opcodes that is not seen before. This might contribute to noise in the results. Hence, during scoring whenever HMM encounters an unseen opcode, we skip the opcode. This experiment was performed for mixed case alone for the families Zbot and ZeroAccess. The results obtained are shown in Table 8. The column Experiment 2 in Table 8 has the scores from Section 4.3.2 and column Experiment 3 has the scores obtained from the experiment in this section.

Table 8: AUCs for opcode sequence - Dynamic/Static - Skip unseen opcodes

| Family | Static | Dynamic/Static | |
|---|---|---|---|
| | | Experiment 2 | Experiment 3 |
| Smarthdd | 0.9952 | 1.0000 | 1.0000 |
| ZeroAccess | 0.7760 | 0.5600 | 0.8970 |
| Cleaman | 0.8567 | 0.6889 | 0.7156 |
| Winwebsec | 0.6609 | 0.6064 | 0.7004 |
| Zbot | 0.7755 | 0.6997 | 0.6424 |
| Security Shield | 0.5028 | 0.5752 | 0.6212 |
| Harebot | 0.5289 | 0.5961 | 0.5694 |

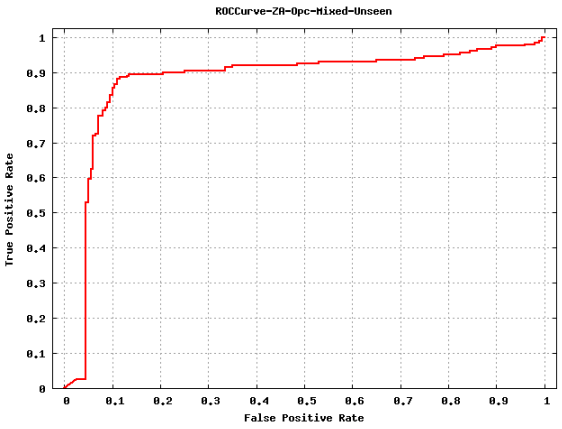


Figure 7: ROC for ZeroAccess-Mixed-Opcodes-Skip unseen opcodes

From the results shown in Table 8, we see that for the ZeroAccess family, removing the unseen opcodes has a good effect on its score. The score has increased from

0.5600 to 0.8970 and also has exceeded the static case with score 0.7760. The ROC curve for ZeroAccess family is shown in Figure 7. The rest of the families, even if they do not have a significant increase in score, families like Winwebsec and Security Shield have improved from the scores obtained in Experiment 2 from Section 4.3.2 and exceed the static case. In each of these experiments, some malware families perform very well and some families don't perform as expected. This inconsistency in results was studied further and led to the next experiment.

### 4.3.4 Experiment 4: Training data from different time windows

The dynamic data generated using IDA was obtained by running each executable for a duration of average 3 to 5 minutes. This duration does not represent the actual duration of the execution rather, the duration IDA Pro takes to log the instructions in the trace log. Since there is no clear way to determine the number of minutes an executable has to run to collect data, and also since most of the malware complete their infection process within the first one minute of their execution, an average 3 to 5 minutes was set for data collection. The same data was used for the dynamic case too and the accuracy of the dynamic case has proved that the data collected is sufficient enough to represent the behavior of a particular malware family to train an HMM. But, in the dynamic/static case, since the model is trained with dynamic data and tested with static data, the data collected might be partially representing the behavior of the family as a whole, and when tested with static data, the gaps might lead to inconsistencies in results. This led to the next experiment, where we run each executable for a longer duration of time. Also, we tested if there is a change in scores for each 10 minute window to see in which time frame does a family of malware exhibit most malicious activity. The results obtained are shown in Table 9.

Table 9: AUCs for opcode sequence - Dynamic/Static - Different time windows

| Family | Dynamic | | Static | Dynamic/Static | | | |
|---|---|---|---|---|---|---|---|
| | 0-3min | 0-40min | - | 0-3min | 0-10min | 10-20min | 0-40min |
| Harebot | 0.7210 | 0.8825 | 0.5289 | 0.5961 | 0.7559 | 0.7894 | 0.7690 |
| Zbot | 0.9676 | 0.9444 | 0.7000 | 0.7216 | 0.7576 | 0.7292 | 0.7600 |
| Cleaman | 1.0000 | 1.0000 | 0.8567 | 0.6889 | 0.6889 | 0.6556 | 0.7056 |

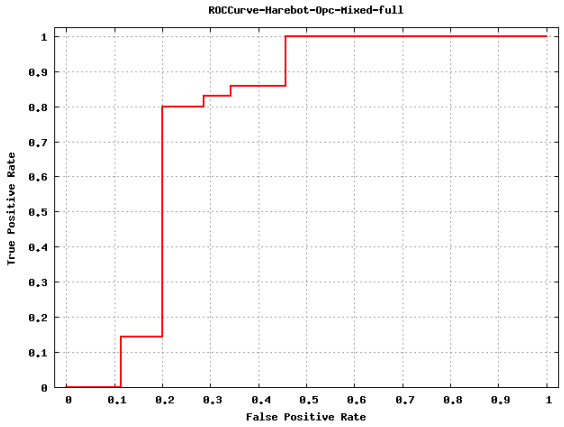


Figure 8: ROC for Harebot-Mixed-Opcodes-0-40min

From the results in Table 9, we see that the scores of Zbot increases from 0.6997 to 0.7600 for the mixed case but does not exceed the static case and decreases from 0.9676 to 0.9444 for the dynamic case. Whereas, for the Harebot family, the score has improved from 0.5961 to 0.7690, has exceeded the static case which is 0.5289 and also has shown considerable improvement for the dynamic case. For the Cleaman family, there is slight increase in scores but does not exceed the static case. The ROC curve for the Harebot family for the mixed case for 0-40 mins is shown in Figure 8.

### 4.3.5 Experiment 5: Static/Dynamic case

For the last experiment, we ran the Static/Dynamic case. In this experiment, instead of training with dynamically obtained opcode sequences, we train using stat-

ically obtained opcode sequences. Similarly, instead of testing with the static data as we have previously seen, we test using the dynamic data. Although not practical, since testing with dynamic data is very costly, this experiment could give some information on the mixed case. The vast difference between the number of distinct opcodes were suspected to cause unexpected scores for the mixed case. The results obtained are shown in Table 10.

Table 10: AUCs for opcode sequence - Static/Dynamic

| Family | Dynamic | Static | Dynamic/Static | Static/Dynamic |
|---|---|---|---|---|
| Smarthdd | 0.9860 | 0.9952 | 1.0000 | 0.9748 |
| Zbot | 0.9681 | 0.7755 | 0.6997 | 0.9525 |
| Cleaman | 1.0000 | 0.8567 | 0.6889 | 1.0000 |
| Winwebsec | 0.8268 | 0.6609 | 0.6064 | 0.6279 |
| Harebot | 0.7210 | 0.5300 | 0.5961 | 0.5832 |
| Security Shield | 0.9452 | 0.5028 | 0.5752 | 0.5928 |
| ZeroAccess | 0.9840 | 0.7760 | 0.5600 | 0.6890 |

From the results shown in Table 10, we can see that Smarthdd, Cleaman and Zbot families show good scores. Zbot family has a score closer to 1.0 and Cleaman family gives the perfect score of 1.0. The AUC curve for Zbot family is shown in Figure 9. Zero Access family also shows increase in scores. The average number of families giving good scores has definitely increased as compared to the Dynamic/Static case. But, the rest of the families such as Security Shield, Winwebsec and Harebot have similar scores to the Dynamic/Static case. This raises the question whether the dynamic and static views are comparable in this manner for all the families.

From the results presented in above sections, we see that a few experiments perform well for a few families but has not given expected results for all families of malware. The distribution of opcodes for the Smarthdd family is very different from

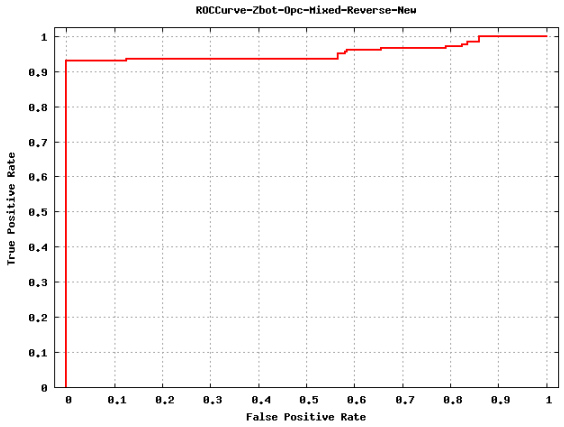


Figure 9: ROC for Zbot-Opcodes-Static/Dynamic

the benign files and hence has given good scores in all experiments. The combined ROC curves for the Smarthdd family is shown in Figure 10. For the ZeroAccess family, lot of unseen opcodes have very small relative frequency but together sufficient enough to cause noise. When the unseen opcodes are removed, the score drastically improved. This can be seen in Figure 11. For the Harebot family, when the executables are run for a longer duration of time, the score has improved. The combined ROC curve for the Harebot family is shown in Figure 12. For the Cleaman and Zbot families, the reverse of the Dynamic/Static case has given good scores. Security Shield and Zbot presented good scores with API call sequences. And the Winwebsec family has shown a slight increase in scores as compared to the static case when the unseen opcodes are removed. In all the above experiments, the scores have slightly increased for various families and in most cases have even exceeded the static case but have not shown a significant increase or a trend that is generic to all families of malware. We discuss the conclusion that we obtain from the experiments performed and possible future work in Chapter 5.

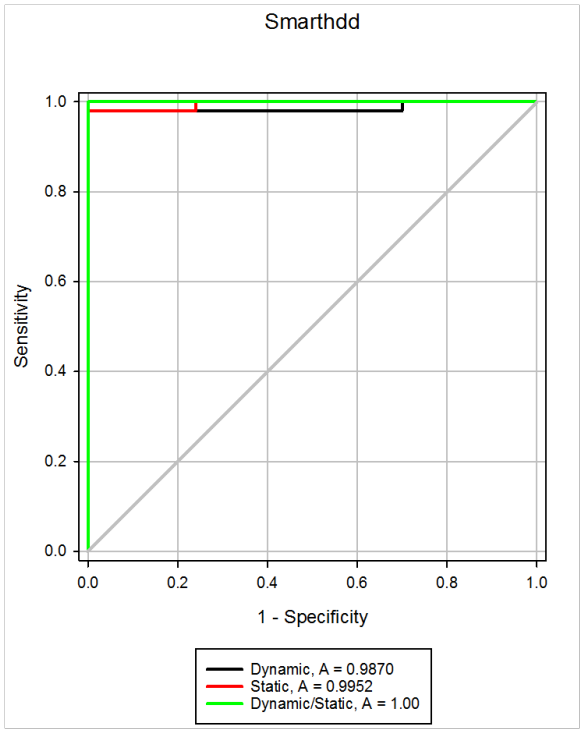


Figure 10: Combined ROC for Smarthdd

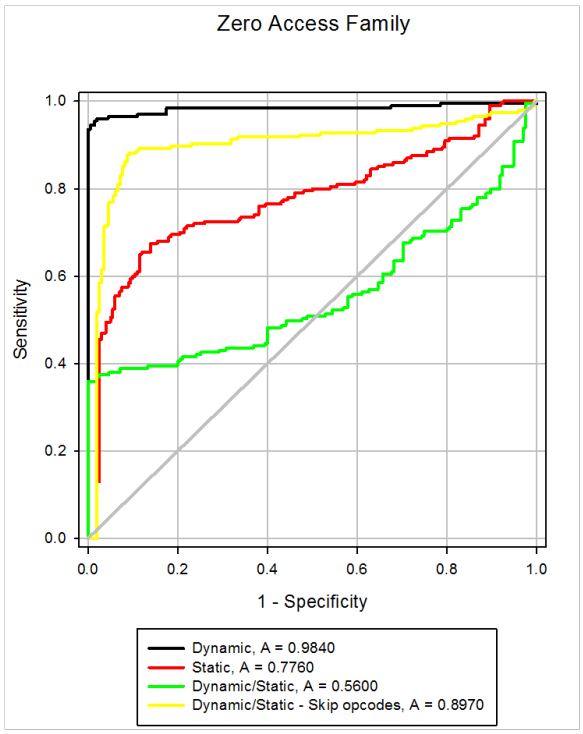


Figure 11: Combined ROC for ZeroAccess-Skip unseen opcodes

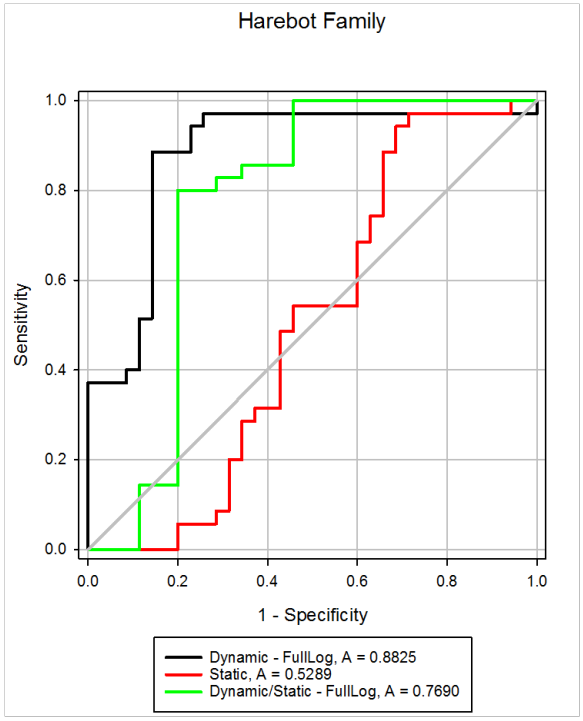


Figure 12: Combined ROC for Harebot-0-40min

# CHAPTER 5

## Conclusion and Future Work

The purpose of this project is to come up with an efficient technique for malware detection that combines the advantages of both dynamic and static analysis. Although, static analysis is simple, fast and practical, using dynamic analysis has proven to give more accurate detection results than static analysis. By combining both the analysis methods, we could get a feasible approach where we could get accurate results with a learning model trained on dynamic data and gain speed by using only static data for scoring.

From all the experiments performed in this project, we conclude that this approach has a lot more to be explored. There have been very good results for a few families in each of the experiments but has not shown consistent results for all families of malware. The reason might be because the dynamic and static views are very different from each other and hence are incomparable the way it is being compared. Also, using an Hidden Markov Model for training and testing with different data for malware detection has not been attempted in the past.

For future work, a perfected approach with this base idea could be very helpful in malware detection. We could combine the feature sets and apply SVM technique for enhanced scores. Or we could focus on only the set of most frequently occurring(MFO) opcodes or group the opcodes into similar sets to compensate the difference in opcodes between static and dynamic views. We could try a variety of other machine learning algorithms with different feature sets using the same base idea.

Since the learning model focuses on dynamic data which strips away one layer of

obfuscation, the malicious activity is accurately captured by the trained model. New malware variants of the same family with lots of code obfuscation techniques when scored against this trained model is expected to stand out against a benign set which will ensure a good detection accuracy.

## LIST OF REFERENCES


[1] F. Ahmed, H. Hameed, M. Zubair Shaq, and M. Farooq. Using spatio-temporal information in API calls with machine learning algorithms for malware detection. *ACM Workshop on Security and Artificial Intelligence* (2009)

[2] B. Anderson, et al. Graph-based Malware Detection using Dynamic Analysis. *Journal of Computer Virology*, Volume 7, Issue 4, pp. 247–258 (2011)

[3] C. Annachhatre, T. H. Austin, M. Stamp. Hidden Markov Models for malware classification. *Journal of Computer Virology and Hacking Techniques*, Springer-Verlag, France (2014)

[4] API Monitor `http://www.rohitab.com/apimonitor`

[5] S. Attaluri, S. McGhee, and M. Stamp. Profile Hidden Markov Models and Metamorphic Virus Detection. *Journal in Computer Virology*, Volume 5, Issue 2, pp. 151–169 (2009)

[6] J. Aycock, Computer Viruses and Malware, *Advances in Information Security*, Springer-Verlag, New York (2006)

[7] A. P. Bradley. The use of the Area Under the ROC Curve in the Evaluation of Machine Learning Algorithms. *Journal Pattern Recognition*, Volume 30, Issue 7, pp. 1145–1159. (1997)

[8] Buster Sandbox Analyser `http://bsa.isoftware.nl/`

[9] Y. H. Choi , B. J. Han, B. C. Bae, H. G. Oh, et al. Toward extracting malware features for classification using static and dynamic analysis. *Computing and Networking Technology (ICCNT)* (2012)

[10] M. Christodorescu, S. Jha. Static analysis of executables to detect malicious patterns. *USENIX Security Symposium* (2003)

[11] Collaborative RCE Tool Library `http://www.woodmann.com/collaborative/tools/index.php/Category:RCE_Tools`

[12] J. Dai, R. Guha, J.Lee. Efficient Virus Detection Using Dynamic Instruction Sequences. *Journal of Computers*, Volume 4, Issue 5, pp. 405–414 (2009)

[13] E. Daoud, I. Jebril. Computer Virus Strategies and Detection Methods. *International Journal of Open Problems in Computer Science and Mathematics*, Volume 1, Issue 2. `http://www.emis.de/journals/IJOPCM/files/IJOPCM(vol.1.2.3.S.08).pdf` (2008)

[14] P. Deshpande, Metamorphic Detection Using Function Call Graph Analysis (2013). Master's Projects. Paper 336. http://scholarworks.sjsu.edu/etd_projects/336

[15] A. Dinaburg, P. Royal, M. Sharif, W. Lee. Ether: Malware Analysis via Hardware Virtualization Extensions. *CCSâĂŹ08, October 27âĂŞ31, 2008, Alexandria, Virginia, USA.* http://ether.gtisc.gatech.edu/ether_ccs_2008.pdf

[16] M. Egele, T. Scholte, E. Kirda, C. Kruegel. A Survey on Automated Dynamic Malware Analysis Techniques and Tools, *Journal ACM Computing Surveys*, Volume 44, Issue 2, Article No. 6 (2012)

[17] M. Eskandari, S. Hashemi. A graph mining approach for detecting unknown malwares. *Journal of Visual Languages and Computing*, Volume 23, Issue 3, pp. 154–162 (2012)

[18] M. Eskandari, Z. Khorshidpour, S. Hashemi. HDM-Analyser: a hybrid analysis approach based on data mining techniques for malware detection. *Journal of Computer Virology and Hacking Techniques*, Volume 9, Issue 2, pp. 77–93 (2013)

[19] M. Eskandari, Z. Khorshidpur, S. Hashemi. To Incorporate Sequential Dynamic Features in Malware Detection Engines. *Intelligence and Security Informatics Conference (EISIC), 2012 European* pp. 46–52 (2012)

[20] R. Fasikhov, API logger tool. http://blackninja2000.narod.ru/rus/api_logger.html

[21] T. Fawcett. An Introduction to ROC Analysis. http://people.inf.elte.hu/kiss/13dwhdm/roc.pdf

[22] Z. Ghahramani. An Introduction to Hidden Markov Models and Bayesian Networks. *International Journal of Pattern Recognition and Artificial Intelligence*, Volume 15, Issue 1, pp. 9–42 (2001)

[23] M. Ghiasi, A. Sami, Z. Salehi. Dynamic malware detection using registers values set analysis.*Information Security and Cryptology (ISCISC)*, pp. 54-59 (2012)

[24] Harebot.M http://www.pandasecurity.com/homeusers/security-info/220319/Harebot.M

[25] G. Jacob, H. Debar, E. Filiol. Behavioral detection of malware: from a survey towards an established taxonomy. *Journal of Computer Virology*, Volume 4, Issue 3, pp. 251–266 (2008)

[26] C. Kolbitsch, P. M. Comparetti, C. Kruegel, E. Kirda, X. Zhou, and X. Wang. Effective and efficient malware detection at the end host. *USENIX Security Symposium* (2009)

[27] D. Lin, M. Stamp. Hunting for Undetectable Metamorphic Viruses. *Journal in Computer Virology*, Volume 7, Issue 3, pp. 201–214 (2011)

[28] S. Momina Tabish. A Graph Theoretic Approach to Malware Detection, Master's Project. Paper 3A84. `http://http://etd.lib.msu.edu/islandora/object/etd` (2012)

[29] A. Moser, C. Kruegel, E. Kirda. Limits of Static Analysis for Malware Detection, *Computer Security Applications Conference*, pp. 421–430 (2007)

[30] Y. Park, D. Reeves, V. Mulukutla and B. Sundaravel, Fast malware classification by automated behavioral graph matching. *Proceedings of the 6th Annual Workshop on Cyber Security and Information Intelligence Research* (2010)

[31] Y. Park, D. Reeves, M. Stamp, Deriving common malware behavior through graph clustering. *Computers and security*, Volume 39, pp. 419–430 (2013)

[32] S. Priyadarshi, Metamorphic Detection via Emulation, Master's Projects, Paper 177 `http://scholarworks.sjsu.edu/etd_projects/177`

[33] Y. Qiao, J. He, Y. Yang , L. Ji. Analyzing Malware by Abstracting the Frequent Itemsets in API Call Sequences. *Trust, Security and Privacy in Computing and Communications (TrustCom)*, pp. 265–270 (2013)

[34] B. Rad, M. Masrom, S. Ibrahim. Camouflage in Malware: from Encryption to Metamorphism. *International Journal of Computer Science and Network Security*, Volume 12, Issue 8, pp. 74 (2012)

[35] J. Rhee, R. Riley, D. Xu and X. Jiang. Kernel malware analysis with un-tampered and temporal views of dynamic kernel memory. *Recent Advances in Intrusion Detection*, Issue 6307, pp. 178–197 (2010)

[36] K. A. Roundy , B. P. Miller, Hybrid analysis and control of malware, *RAID'10 Proceedings of the 13th international conference on Recent Advances in Intrusion Detection*, Issue September 15-17 (2010)

[37] N. Runwal, R. M. Low, and M. Stamp. Opcode Graph Similarity and Metamorphic Detection. *Journal in Computer Virology*, Volume 8, Issue 1-2, pp. 37–52 (2012)

[38] SandBoxie, `http://sandboxie.com/`

[39] Security Shield, `http://www.symantec.com/security_response/glossary/define.jsp?letter=s&word=security-shield`

[40] M. K. Shankarapani, S. Ramamoorthy, R. S. Movva, S. Mukkamala. Malware detection using assembly and API call sequences. *Journal of Computer Virology*, Volume 2, Issue 7, pp. 107–119 (2011)

[41] G. Shanmugam, R. Low, M. Stamp. Simple Substitution Distance and Metamorphic Detection. *Journal of Computer Virology and Hacking Techniques*, Volume 9, Issue 3, pp. 159–170 (2013)

[42] S.M.A.R.T HDD `http://support.kaspersky.com/viruses/rogue?qid=208286454`

[43] M. Stamp. A Revealing Introduction to Hidden Markov Models. `http://www.cs.sjsu.edu/~stamp/RUA/HMM.pdf` (2012)

[44] M. Stamp, Information Security: Principles and Practice, Second Edition, Wiley. (2011)

[45] Symantec White Paper, Internet Security Report, Volume 20 `http://www.symantec.com/security_response/publications/threatreport.jsp` (2015)

[46] Trojan.Cleaman `http://www.symantec.com/security_response/writeup.jsp?docid=2012-050103-5441-99`

[47] Trojan.Zbot `http://www.symantec.com/security_response/writeup.jsp?docid=2010-011016-3514-99`

[48] Trojan.ZeroAccess `http://www.symantec.com/security_response/writeup.jsp?docid=2011-071314-0410-99`

[49] A. Walenstein, R. Mathur, M. Chouchane, R. Chouchane, and A. Lakhotia. The Design Space of Metamorphic Malware. *In Proceedings of the 2nd International Conference on Information Warfare* (2007)

[50] WinAPIOverride `http://jacquelin.potier.free.fr/winapioverride32/`

[51] Win32/Winwebsec `http://www.microsoft.com/security/portal/threat/encyclopedia/entry.aspx?Name=Win32%2fWinwebsec`

[52] W. Wong. Analysis and Detection of Metamorphic Computer Viruses. Master's Projects. Paper 153. `http://scholarworks.sjsu.edu/etd_projects/153` (2006)

[53] Y. Ye, D. Wang, T. Li, D. Ye, Q. Jiang An intelligent PE-malware detection system based on association mining. *Journal of Computer Virology*, Volume 4, Issue 4, pp. 323–334 (2008)

# APPENDIX A

## Appendix A: ROC curves

### A.1 Using API call sequence

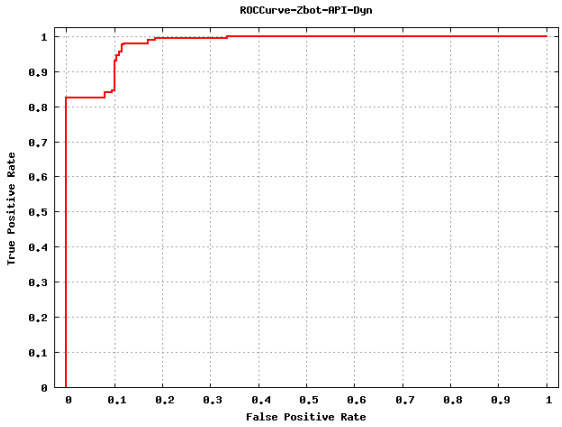


Figure A.13: ROC for Zbot-Dynamic-API

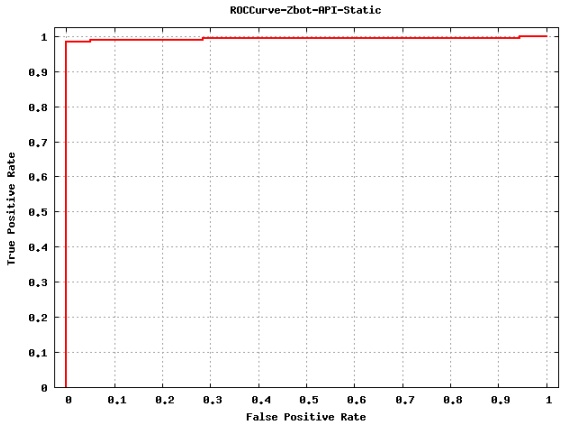


Figure A.14: ROC for Zbot-Static-API

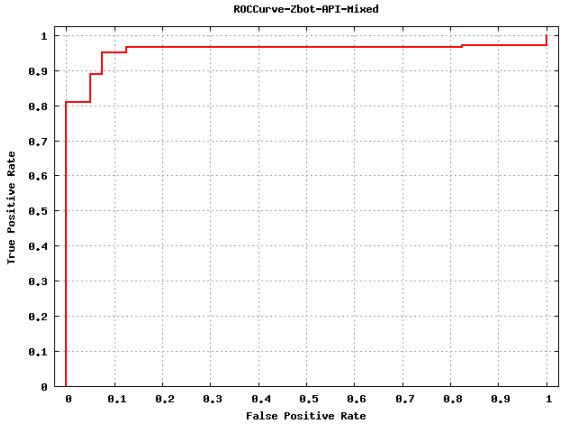


Figure A.15: ROC for Zbot-Mixed-API

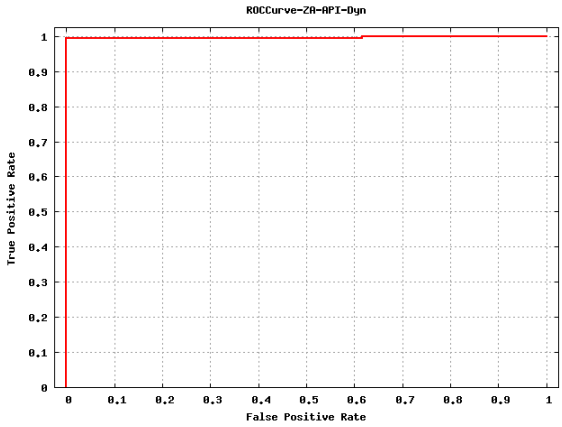


Figure A.16: ROC for ZeroAccess-Dynamic-API

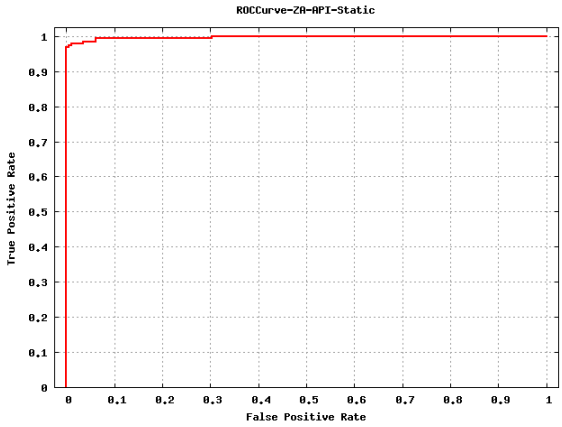


Figure A.17: ROC for ZeroAccess-Static-API

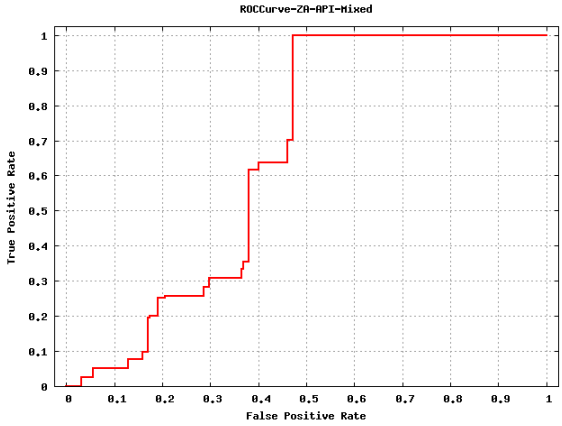


Figure A.18: ROC for ZeroAccess-Mixed-API

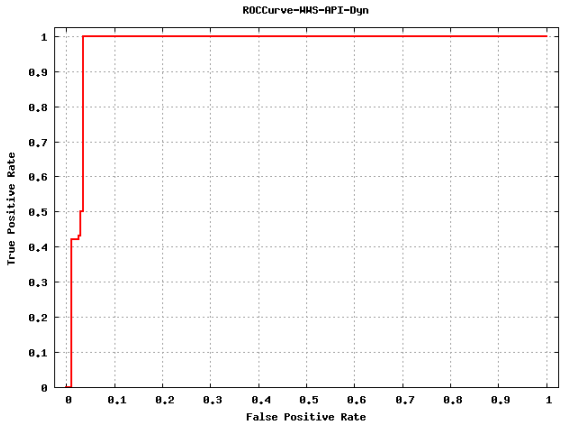


Figure A.19: ROC for Winwebsec-Dynamic-API

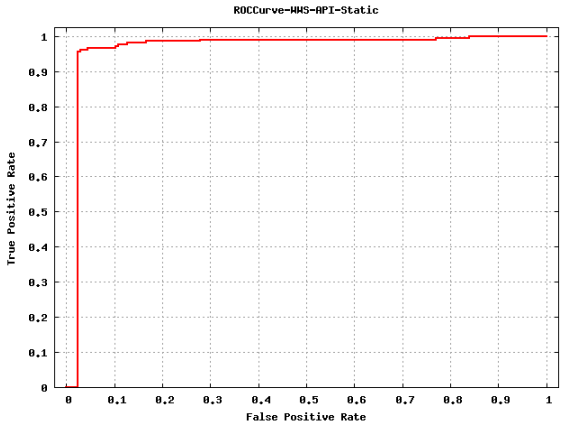


Figure A.20: ROC for Winwebsec-Static-API

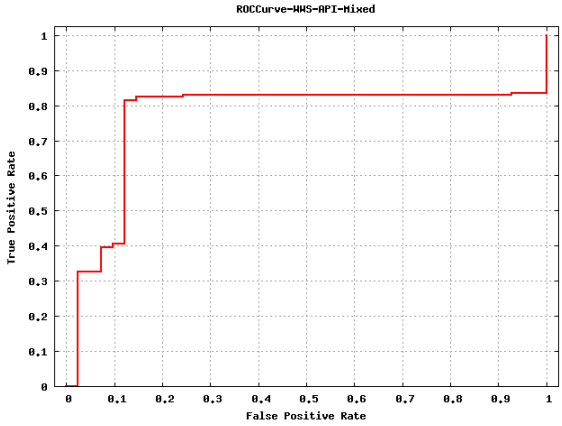


Figure A.21: ROC for Winwebsec-Mixed-API

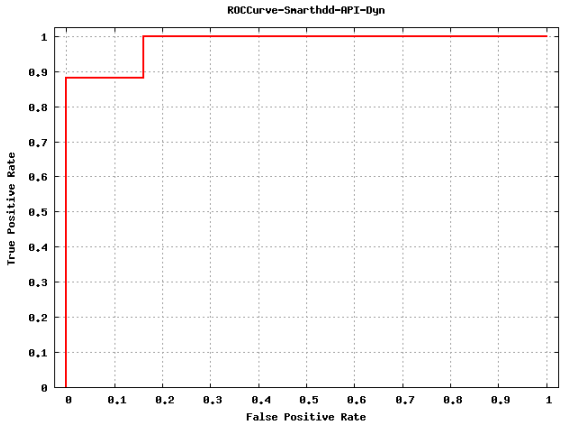


Figure A.22: ROC for Smarthdd-Dynamic-API

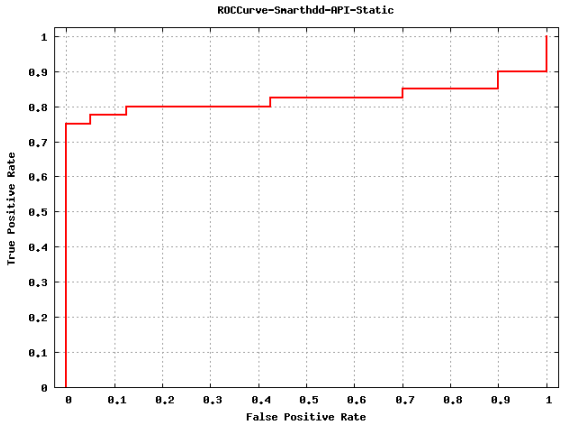


Figure A.23: ROC for Smarthdd-Static-API

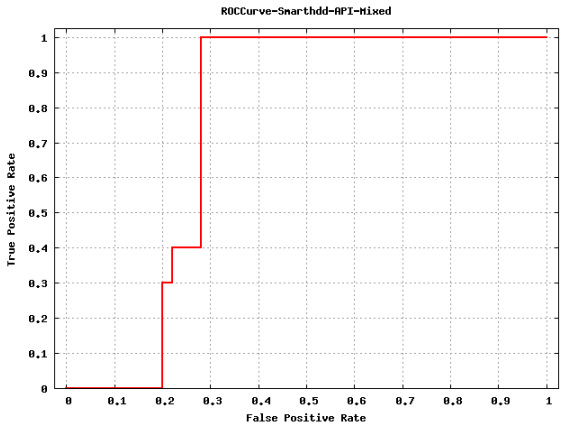


Figure A.24: ROC for Smarthdd-Mixed-API

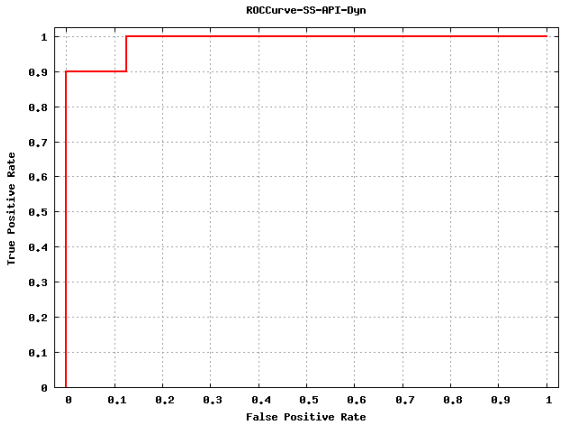


Figure A.25: ROC for Security Shield-Dynamic-API

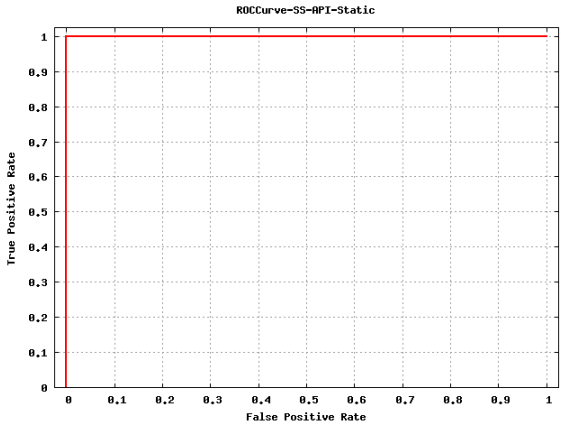


Figure A.26: ROC for Security Shield-Static-API

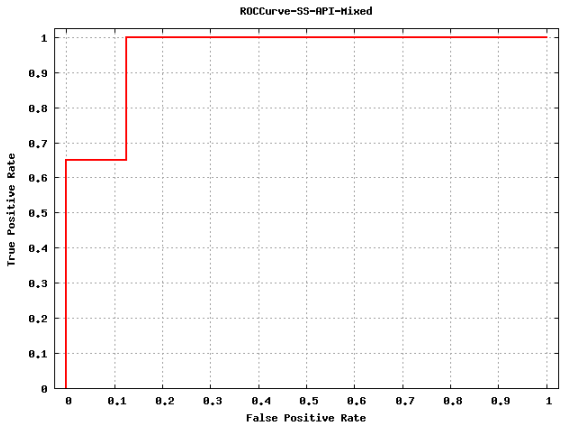


Figure A.27: ROC for Security Shield-Mixed-API

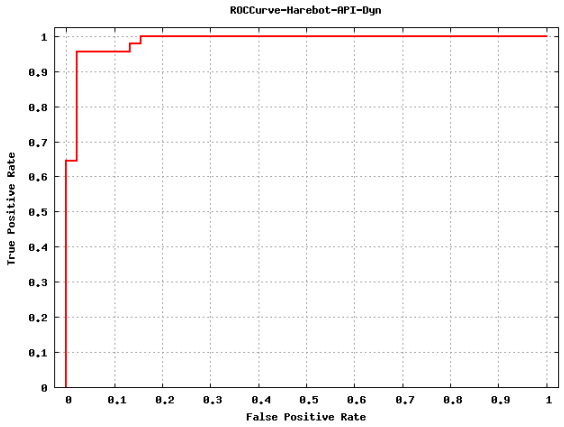


Figure A.28: ROC for Harebot-Dynamic-API

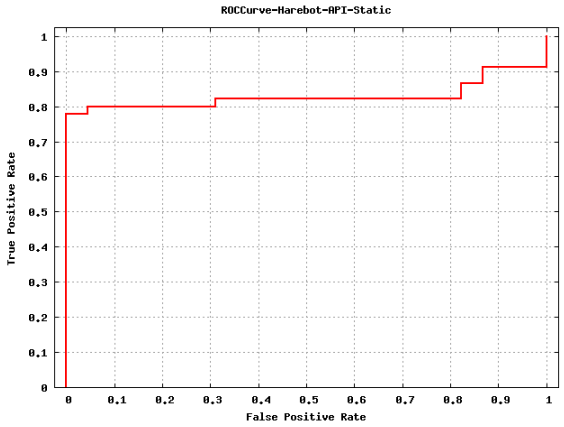


Figure A.29: ROC for Harebot-Static-API

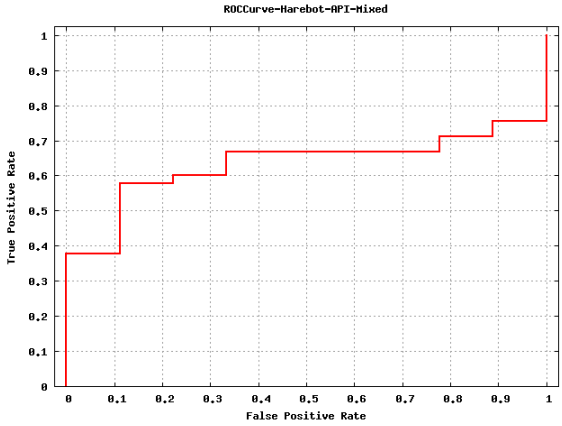


Figure A.30: ROC for Harebot-Mixed-API

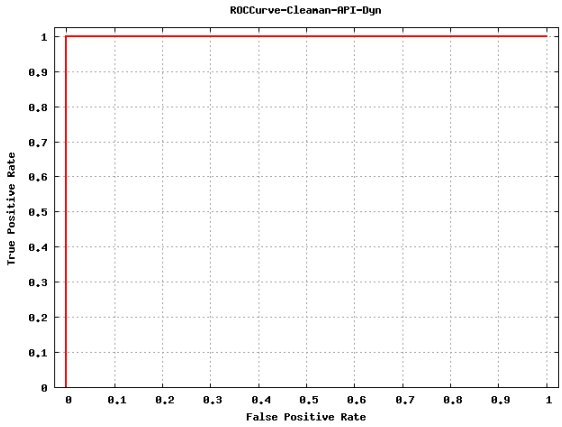


Figure A.31: ROC for Cleaman-Dynamic-API

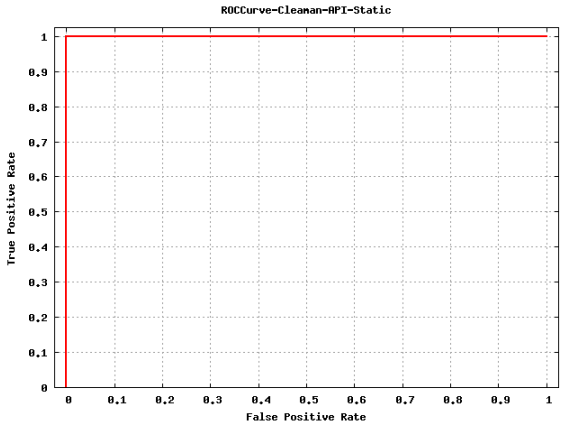


Figure A.32: ROC for Cleaman-Static-API

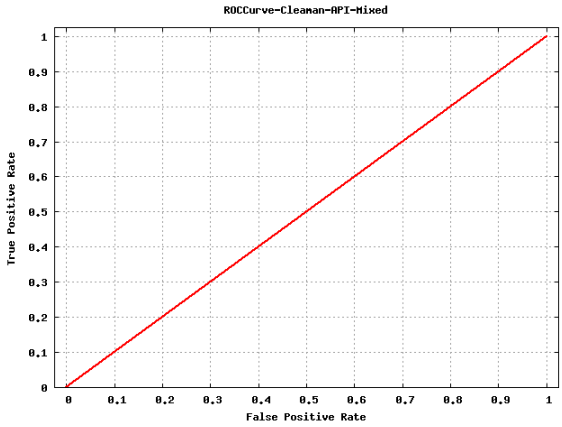


Figure A.33: ROC for Cleaman-Mixed-API

# APPENDIX B

## Appendix B: ROC curves

### B.1 Using opcode sequence - Ether/IDA

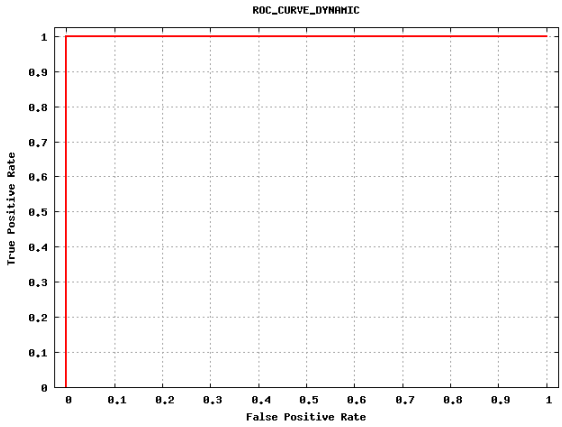


Figure B.34: ROC for Zbot-Dynamic-Ether/IDA-Opcodes

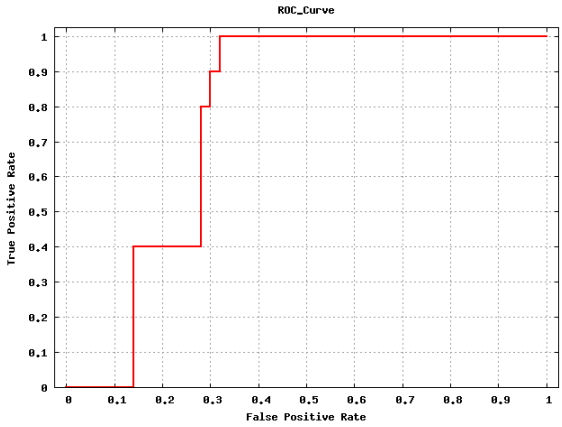


Figure B.35: ROC for Zbot-Mixed-Ether/IDA-Opcodes

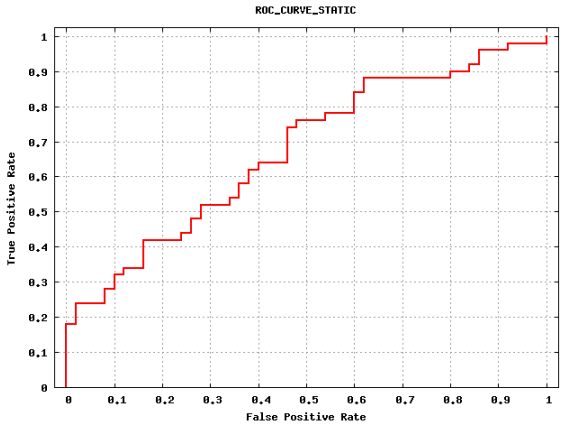


Figure B.36: ROC for Zbot-Static-Ether/IDA-Opcodes

## B.2 Using opcode sequence - IDA/IDA - Small Subset

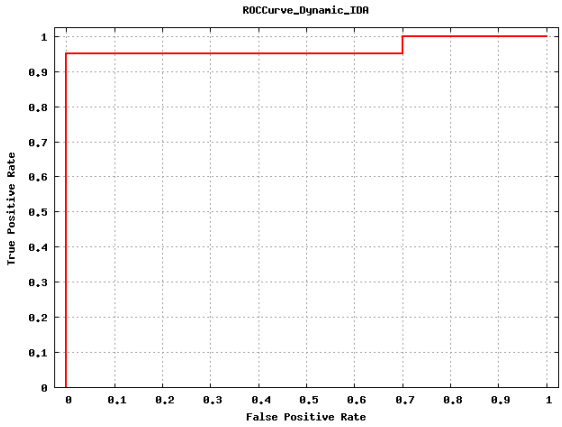


Figure B.37: ROC for Zbot-Dynamic-IDA/IDA-Opcodes

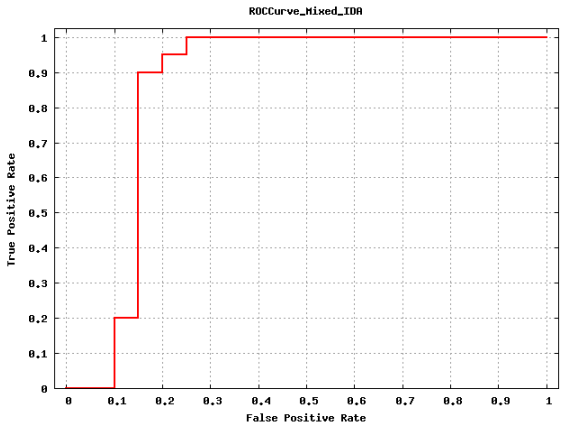


Figure B.38: ROC for Zbot-Mixed-IDA/IDA-Opcodes

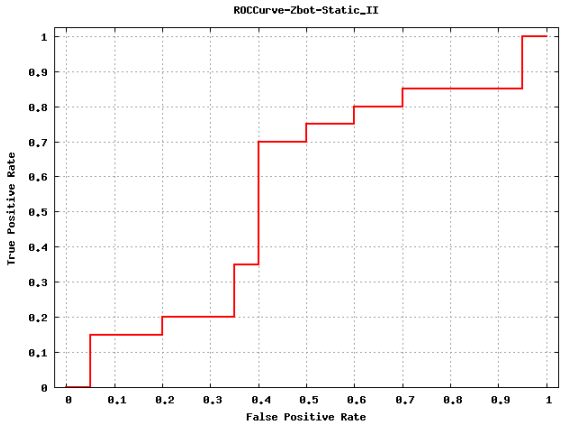


Figure B.39: ROC for Zbot-Static-IDA/IDA-Opcodes

## B.3 Using opcode sequence - IDA/IDA

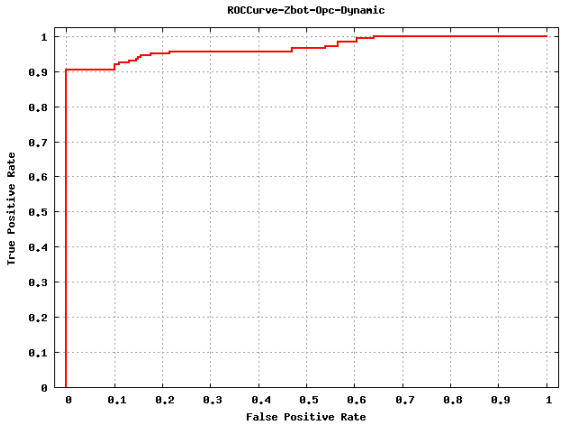


Figure B.40: ROC for Zbot-Dynamic-Opcodes

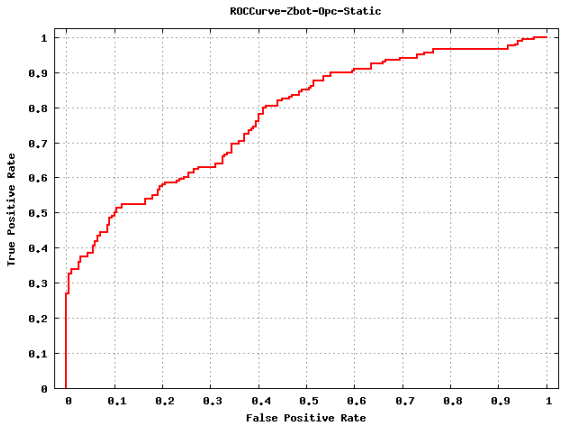


Figure B.41: ROC for Zbot-Static-Opcodes

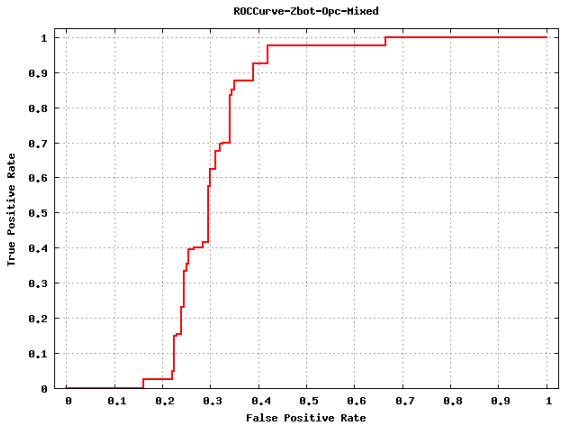


Figure B.42: ROC for Zbot-Mixed-Opcodes

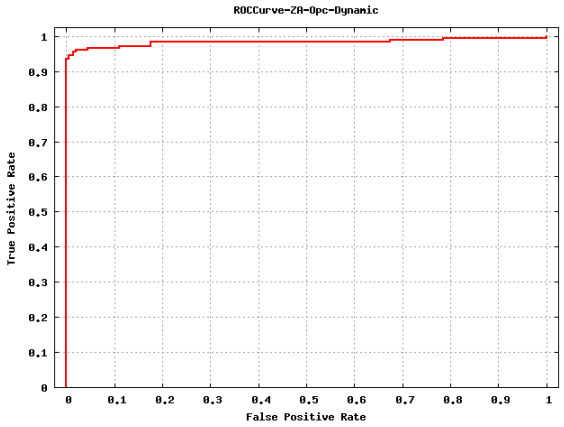


Figure B.43: ROC for ZeroAccess-Dynamic-Opcodes

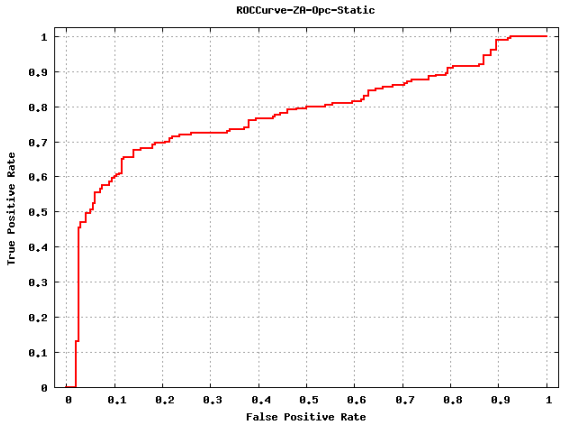


Figure B.44: ROC for ZeroAccess-Static-Opcodes

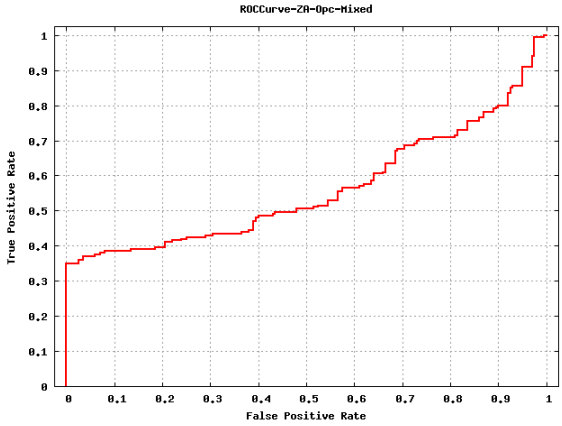


Figure B.45: ROC for ZeroAccess-Mixed-Opcodes

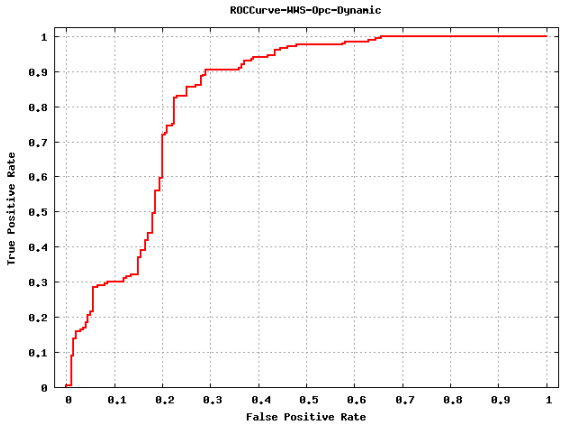


Figure B.46: ROC for Winwebsec-Dynamic-Opcodes

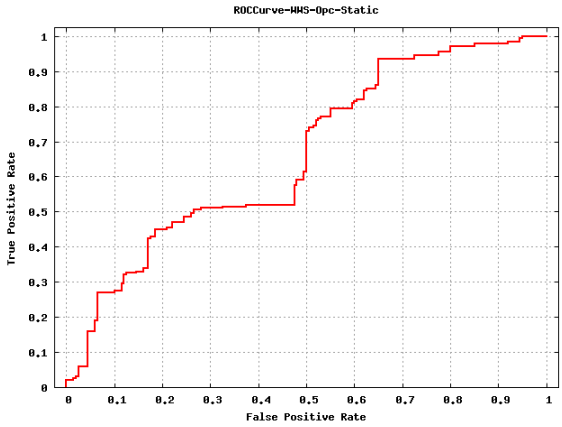


Figure B.47: ROC for Winwebsec-Static-Opcodes

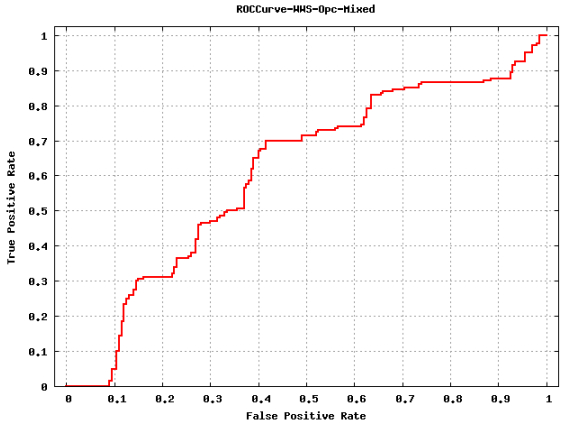


Figure B.48: ROC for Winwebsec-Mixed-Opcodes

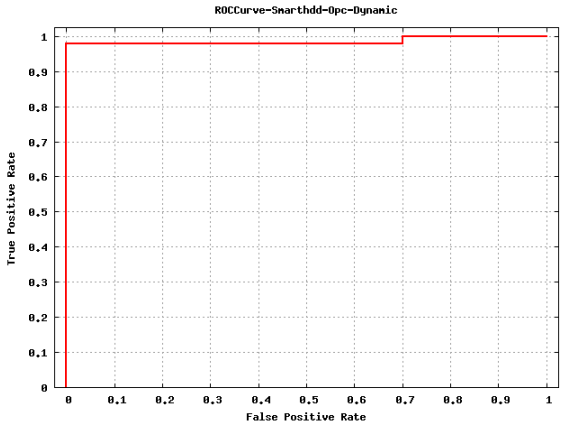


Figure B.49: ROC for Smarthdd-Dynamic-Opcodes

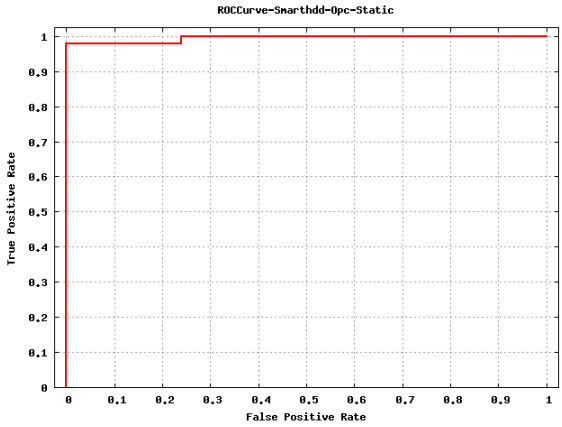


Figure B.50: ROC for Smarthdd-Static-Opcodes

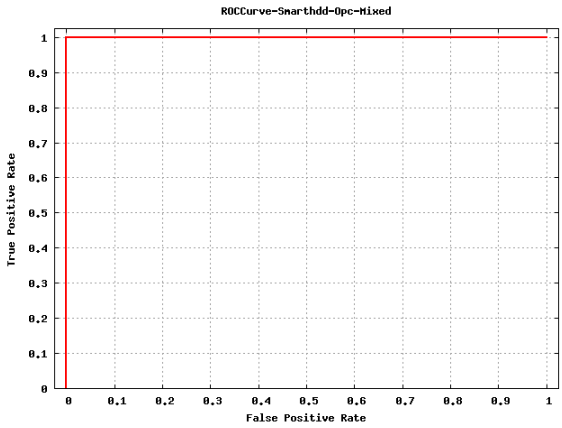


Figure B.51: ROC for Smarthdd-Mixed-Opcodes

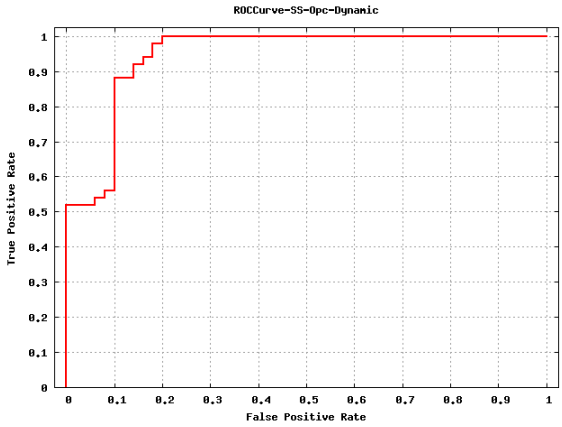


Figure B.52: ROC for Security Shield-Dynamic-Opcodes

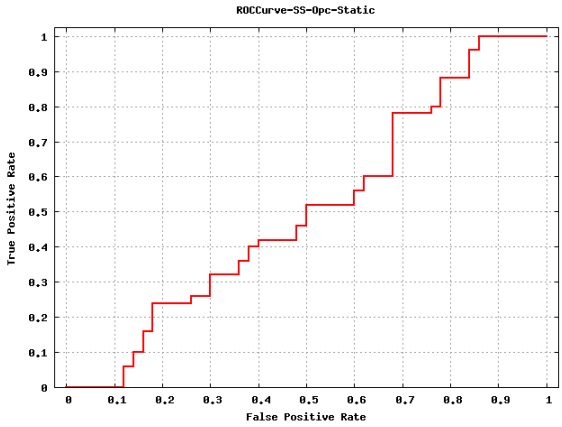


Figure B.53: ROC for Security Shield-Static-Opcodes

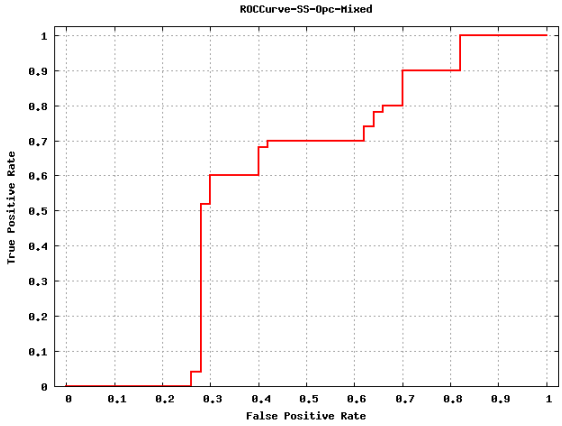


Figure B.54: ROC for Security Shield-Mixed-Opcodes

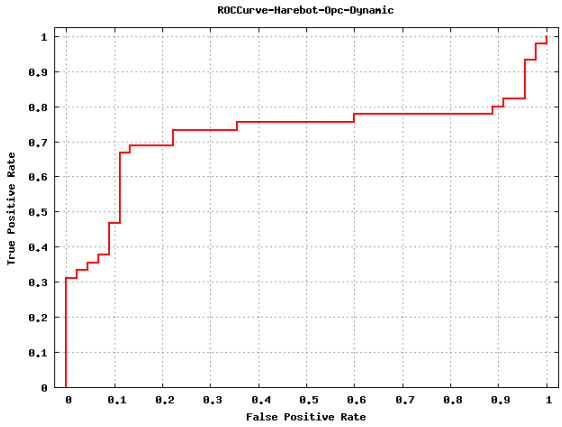


Figure B.55: ROC for Harebot-Dynamic-Opcodes

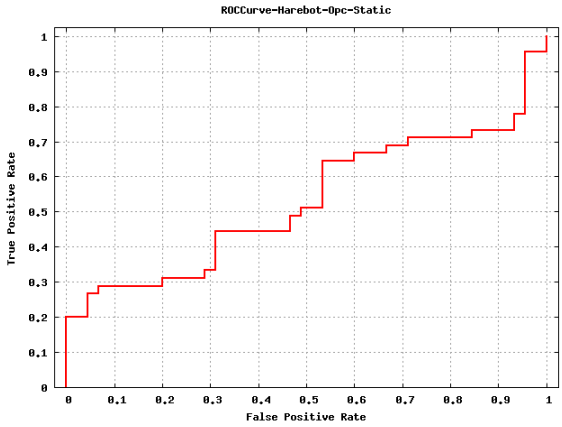


Figure B.56: ROC for Harebot-Static-Opcodes

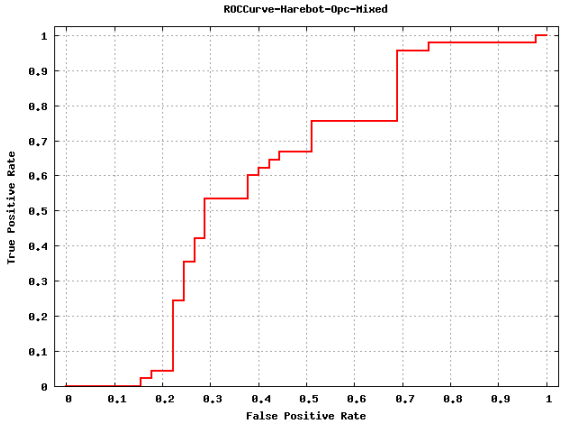


Figure B.57: ROC for Harebot-Mixed-Opcodes

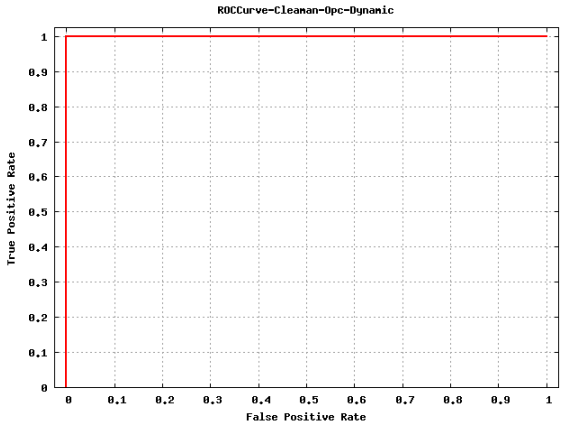


Figure B.58: ROC for Cleaman-Dynamic-Opcodes

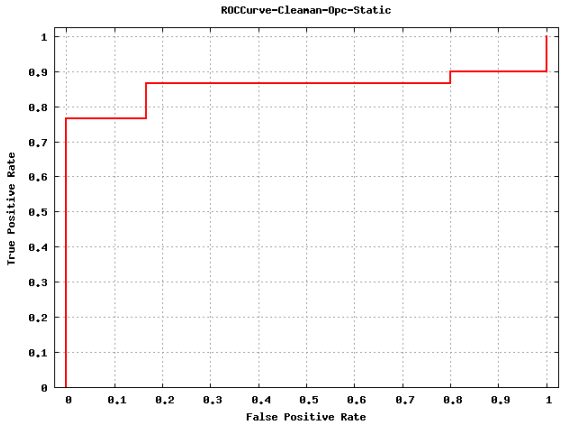


Figure B.59: ROC for Cleaman-Static-Opcodes

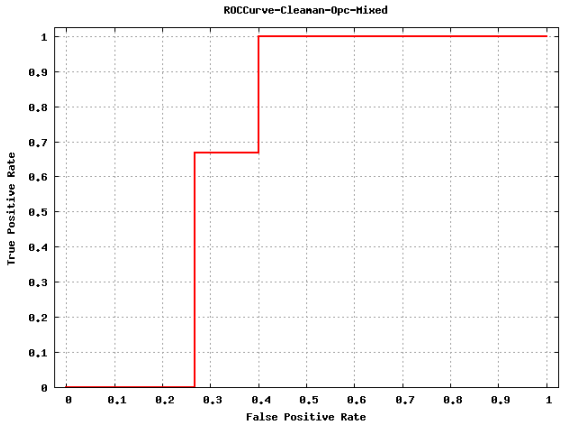


Figure B.60: ROC for Cleaman-Mixed-Opcodes

## B.4 Using opcode sequence - IDA/IDA - Skip unseen opcodes

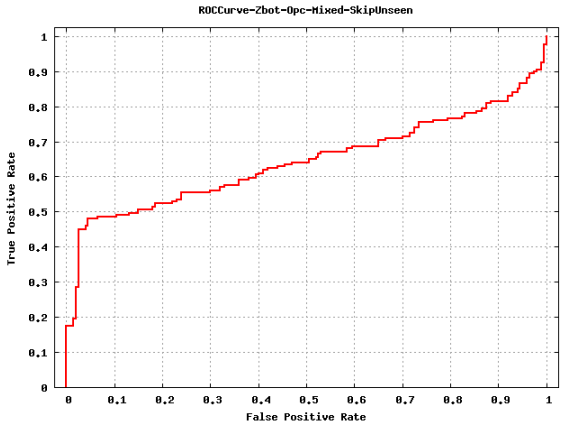


Figure B.61: ROC for Zbot-Mixed-Opcodes-Skip unseen opcodes

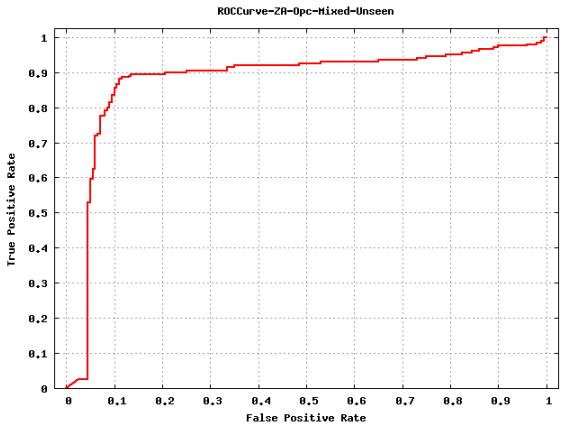


Figure B.62: ROC for ZeroAccess-Mixed-Opcodes-Skip unseen opcodes

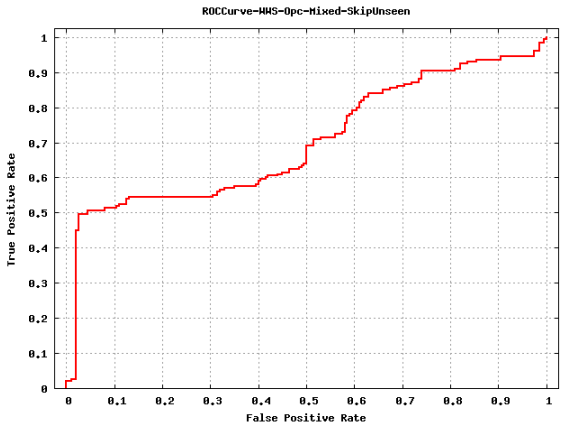


Figure B.63: ROC for Winwebsec-Mixed-Opcodes-Skip unseen opcodes

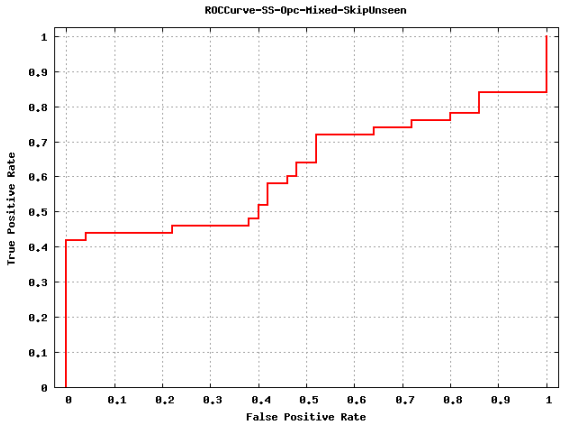


Figure B.64: ROC for Security Shield-Mixed-Opcodes-Skip unseen opcodes

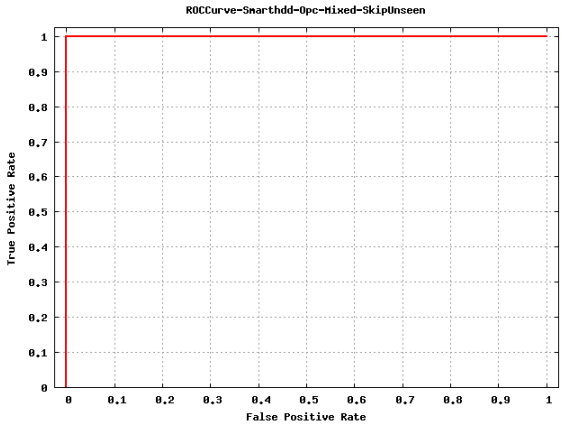


Figure B.65: ROC for Smarthdd-Mixed-Opcodes-Skip unseen opcodes

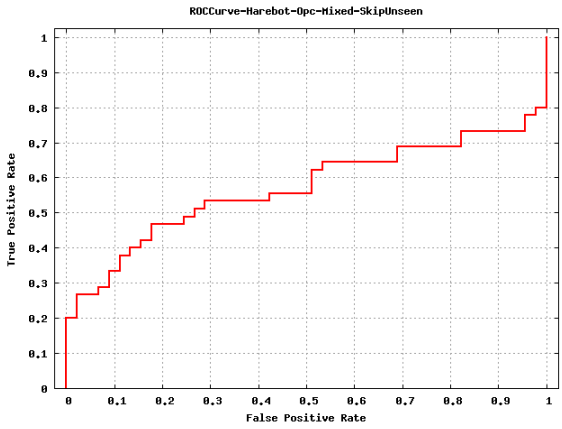


Figure B.66: ROC for Harebot-Mixed-Opcodes-Skip unseen opcodes

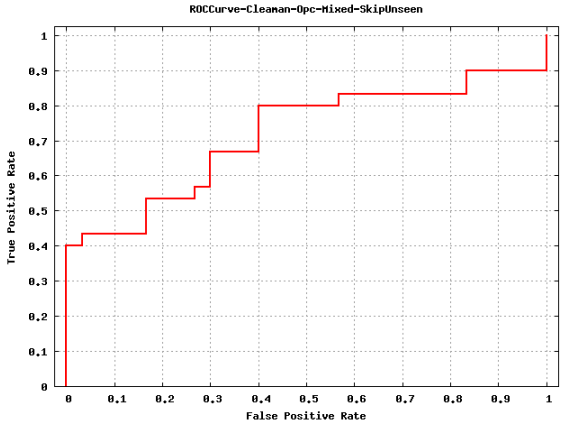


Figure B.67: ROC for Cleaman-Mixed-Opcodes-Skip unseen opcodes

## B.5 Using opcode sequence - IDA/IDA - Different time windows

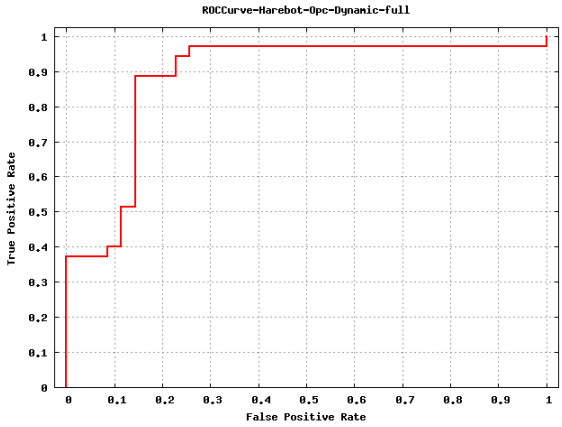


Figure B.68: ROC for Harebot-Dynamic-Opcodes-0-40min

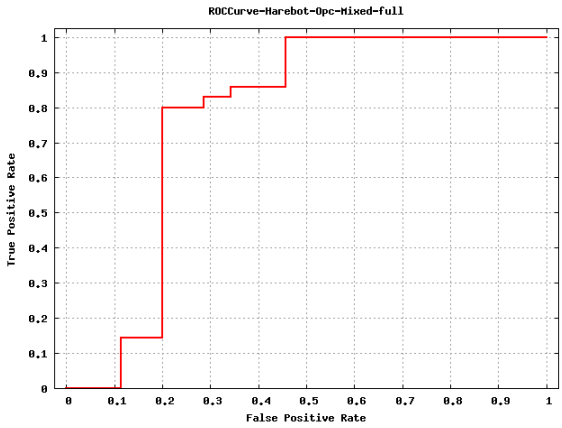


Figure B.69: ROC for Harebot-Mixed-Opcodes-0-40min

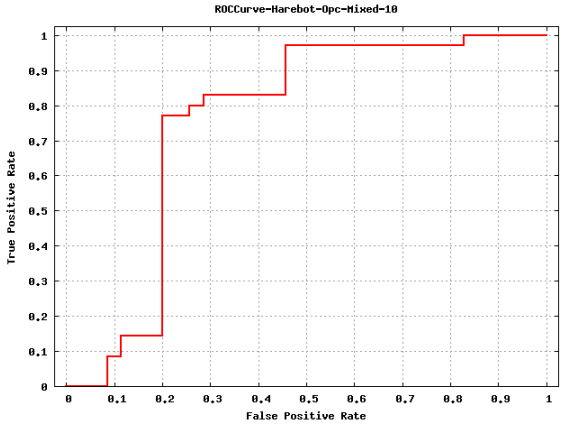


Figure B.70: ROC for Harebot-Mixed-Opcodes-0-10min

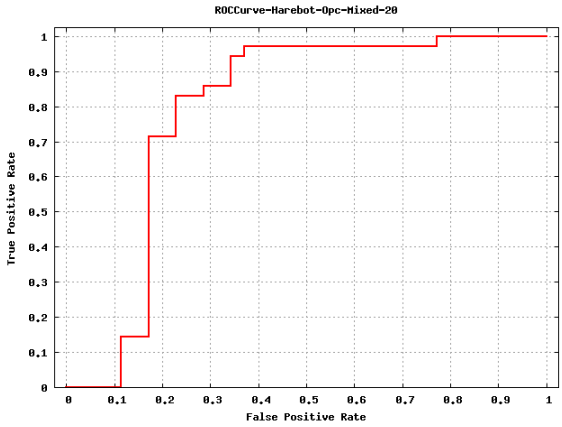


Figure B.71: ROC for Harebot-Mixed-Opcodes-10-20min

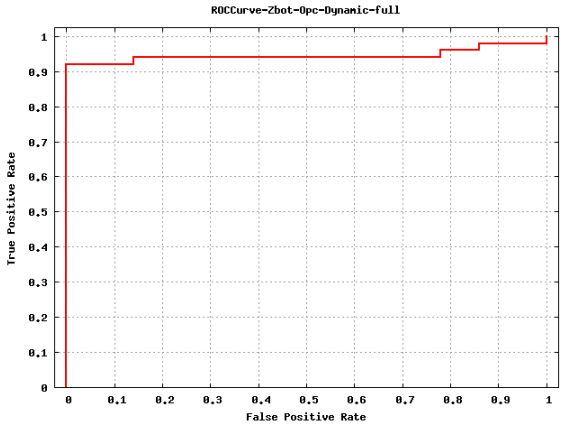


Figure B.72: ROC for Zbot-Dynamic-Opcodes-0-40min

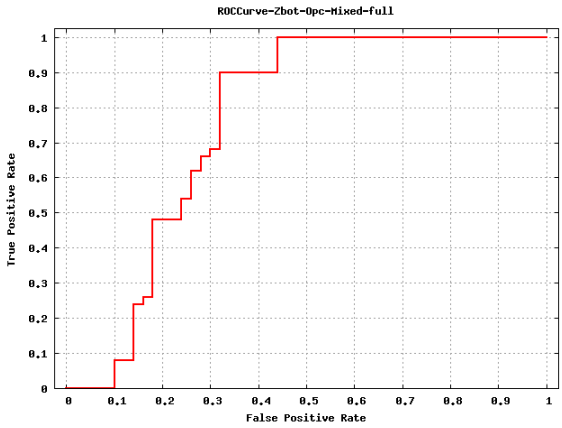


Figure B.73: ROC for Zbot-Mixed-Opcodes-0-40min

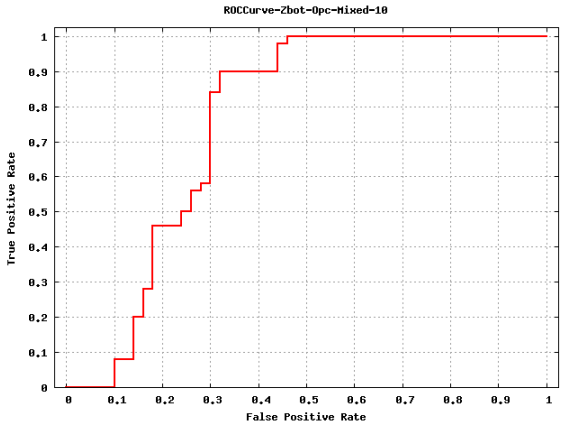


Figure B.74: ROC for Zbot-Mixed-Opcodes-0-10min

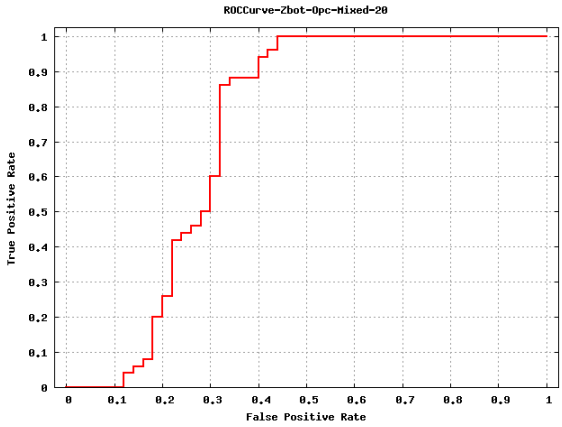


Figure B.75: ROC for Zbot-Mixed-Opcodes-10-20min

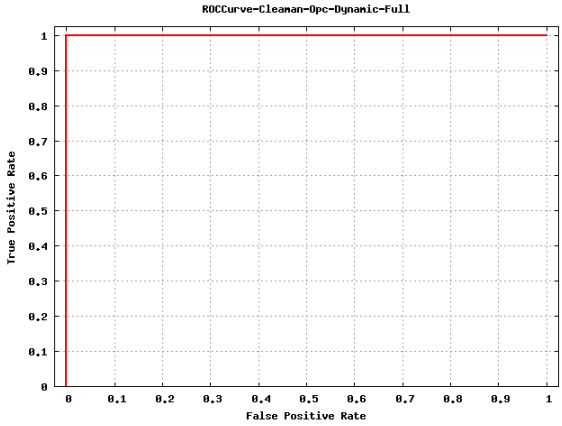


Figure B.76: ROC for Cleaman-Dynamic-Opcodes-0-40min

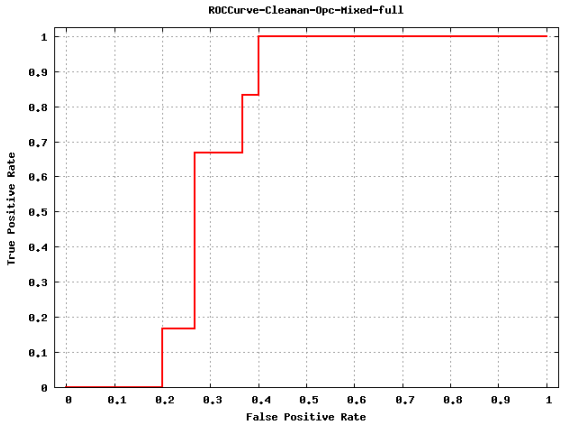


Figure B.77: ROC for Cleaman-Mixed-Opcodes-0-40min

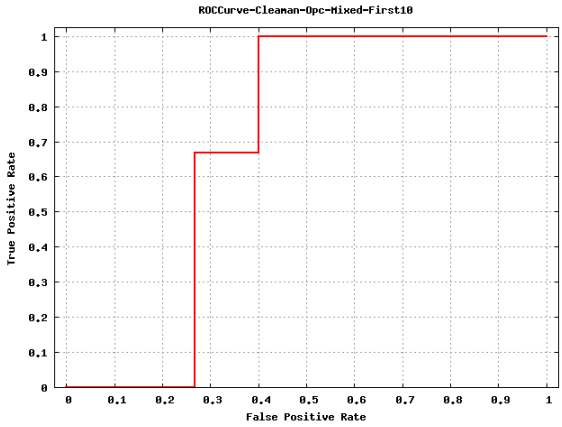


Figure B.78: ROC for Cleaman-Mixed-Opcodes-0-10min

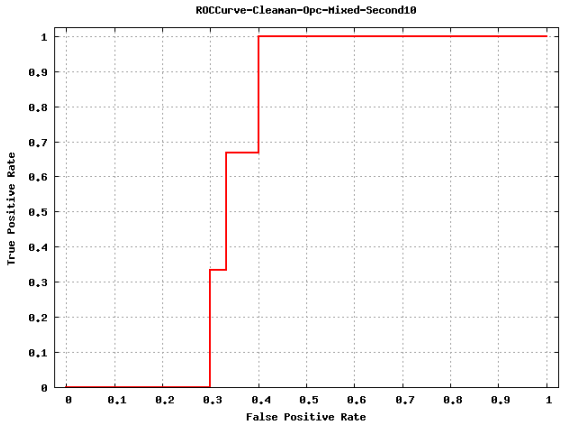


Figure B.79: ROC for Cleaman-Mixed-Opcodes-10-20min

## B.6 Using opcode sequence - IDA/IDA - Static/Dynamic

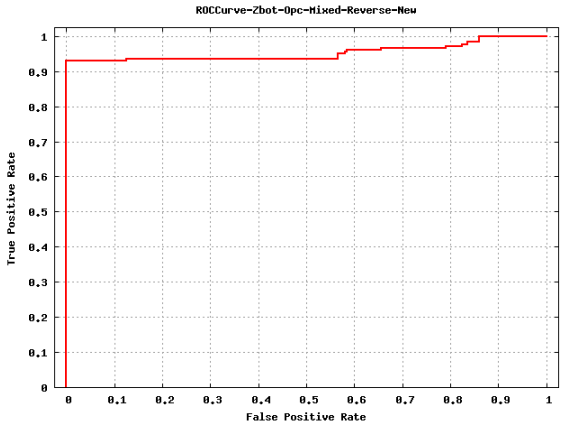


Figure B.80: ROC for Zbot-Mixed-Opcodes-Static/Dynamic

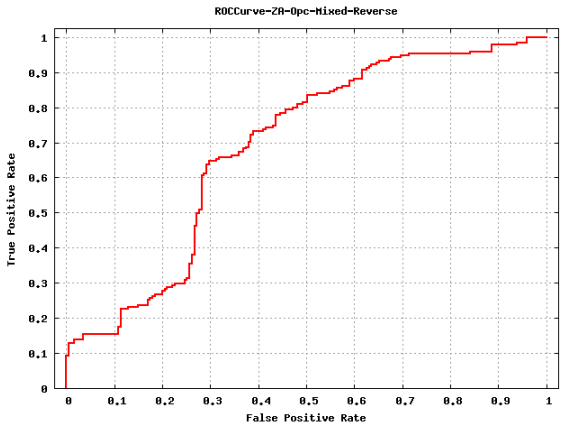


Figure B.81: ROC for ZeroAccess-Mixed-Opcodes-Static/Dynamic

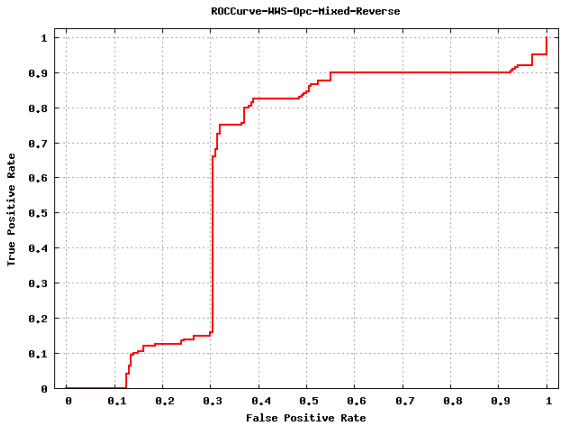


Figure B.82: ROC for Winwebsec-Mixed-Opcodes-Static/Dynamic

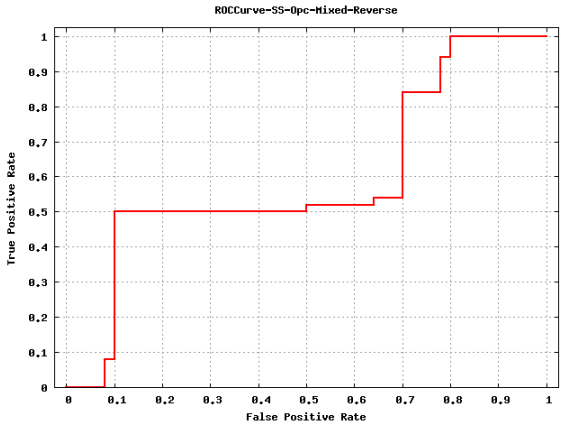


Figure B.83: ROC for Security Shield-Mixed-Opcodes-Static/Dynamic

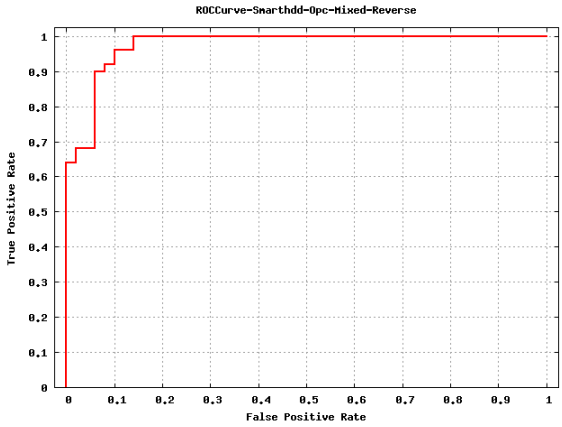


Figure B.84: ROC for Smarthdd-Mixed-Opcodes-Static/Dynamic

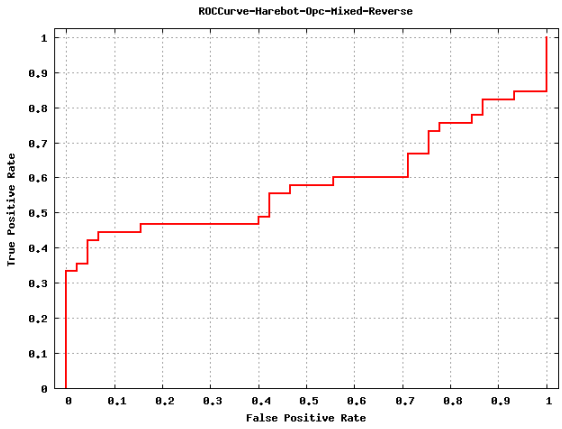


Figure B.85: ROC for Harebot-Mixed-Opcodes-Static/Dynamic

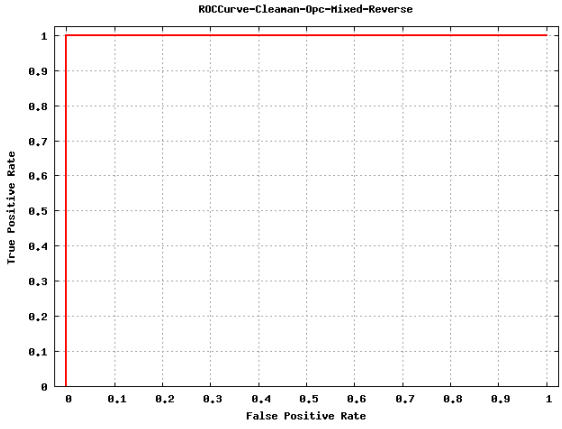


Figure B.86: ROC for Cleaman-Mixed-Opcodes-Static/Dynamic